Practical High-Coverage Sound Predictive Race Detection

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Jake Roemer, B.S.

Graduate Program in Computer Science and Engineering

The Ohio State University

2019

Dissertation Committee:

Michael D. Bond, Advisor
Atanas Rountev
Paul Sivilotti
Abstract

Data races pose a fundamental issue plaguing the reliability of parallel software. A shared-memory program has a data race if two conflicting memory accesses (accesses to the same memory location by different threads, where at least one access is a write) can execute consecutively (no interleaving events exist). Modern shared-memory programming languages including C++ and Java provide undefined or ill-defined semantics for executions with data races. So in the presence of data races, shared-memory programs are vulnerable to fatal crashes, data corruption, and other unexpected errors.

Data races manifest nondeterministically creating the complex task of writing and diagnosing shared-memory programs. Existing research offers a variety of program analyses to detect and report data races to developers. Most notably, predictive dynamic analyses detect data races that can occur in executions other than the observed execution. Predictive analysis defines the set of necessary ordering between events while ensuring the same behavior as the observed execution to detect races that industry standard happens-before analysis cannot detect.

The existing techniques are sound (report only true races) but often miss races because the techniques are impractical to run extensively on large programs in practice due to performing costly reasoning about reordered thread interleavings. Alternative partial-order-based predictive techniques that scale to large programs fail to detect all races knowable from
the observed execution because of the conservative ordering between events that preserve a valid reordered execution for the techniques to be sound.

The partial-order-based predictive techniques remain impractical compared to industry standard race detectors. A major contributing factor to predictive analyses poor performance is the metadata required to weaken the observed ordering. The metadata tracks conflicting critical sections and is a vital component of predictive analysis that industry standard happens-before race detectors do not need to manage. Thus the limitations of predictive analyses result in shared-memory programs remaining vulnerable in the presence of extremely hard-to-detect data races.

The goal of this work is to detect all races knowable from the observed execution and close the performance gap between predictive analysis and industry standard detectors by designing a practical high-coverage sound predictive data race detector. This dissertation presents a body of work that advances data race coverage and run time and memory usage performance of predictive analysis.

The first main focus for this dissertation is to maximize race detection capability to detect all knowable races from an observed execution. Our key insight to achieve high-coverage predictive analysis is to sacrifice soundness (may detect false races) by relinquishing conservative ordering between events. Our contributions reduce ordering between events more than prior work by showing the conservative orderings are unnecessary to ensure the same behavior as the observed execution. Our contributions detect hard-to-find races that are hidden from prior work, thus achieving high-coverage predictive analysis.

To strictly improve race coverage of predictive analysis this dissertation aims to provide the same soundness guarantees as prior predictive analyses and existing industry standard detectors. Our contributions provide soundness (report only true races) by verifying if
potential races detected are true races without sacrificing high-coverage predictive analysis of full program executions.

The second focus for this dissertation is to improve run time and memory usage performance of predictive analysis such that predictive race detectors can compete with industry standard detectors. Our key insight to achieve practical predictive analysis is to apply similar epoch optimizations used by popular happens-before race detectors to conflicting critical section metadata and to be the first to apply the epoch optimizations directly to predictive analysis. Our contributions provide optimizations applicable to several predictive analyses that improve run time and memory usage performance to be competitive with industry standard detectors.

This work tackles real, challenging issues in data race detection that directly impact system reliability. More specifically, advancing predictive analysis not only develops techniques that guarantee higher coverage with reasonable run time and memory usage than prior work but also makes a case for predictive analysis to be the prevailing approach for detecting data races. The insights of this dissertation lend itself to the development of new relations or application of partial orders to other techniques such as static analysis that broadens how an approach detects a data race.
Dedicated to anyone who appreciates their capabilities.
Acknowledgments

Every moment of my life has equally contributed to achieving my Ph.D. The countless individuals, whether in ways that seem insignificant or directly impactful, are credited throughout my endeavors. These people could never be thanked enough since none of what I have accomplished would be possible without each and every one of them.

Specifically, I would like to thank Michael Bond first as an understanding and compassionate friend and second as a supportive advisor. He has taught me more than I deserve and guided me to be the capable researcher I am today. I am eternally grateful for your help.

I want to express my gratitude to the many professors and colleagues that have crossed my path at Ohio State University who supported me and taught me a wealth of knowledge. Especially, thanks to Atanas Rountev, Paul Sivilotti, and Feng Qin for taking the time to serve on my committee, providing encouragement and useful feedback. I must also acknowledge the invaluable collaboration with Yannis Smaragdakis, Steve Freund, Jaeheon Yi, and many others outside of Ohio State University.

I recognize Yufan Xu and the members of the PLaSS group, Swarnendu Biswas, Minjia Zhang, Man Cao, Aritra Sengupta, Rui Zhang, and Kaan Genç that I have had the pleasure to work with. These exceptional colleagues have helped shape ideas through exciting and insightful discussions. I would not trade the time I have spent working with them for anything.
Lastly, I want to thank my family for supporting my decisions and reminding me of my potential throughout my life. To my everlasting friends who have inspired me and etched out a comforting place in this world for me to feel accepted, thank you.
Vita

May 2014 ........................................... B.S.,
                           Physics,
                           Temple University, USA.

Publications


Fields of Study

Major Field: Computer Science and Engineering
Studies in:

<table>
<thead>
<tr>
<th>Area</th>
<th>Professor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming Languages and Software Systems</td>
<td>Prof. Michael D. Bond</td>
</tr>
<tr>
<td>Artificial Intelligence</td>
<td>Prof. John Eric Fosler-Lussier</td>
</tr>
<tr>
<td>High Performance Computing</td>
<td>Prof. Gagan Agrawal</td>
</tr>
</tbody>
</table>
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>v</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>vi</td>
</tr>
<tr>
<td>Vita</td>
<td>viii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xiv</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xvi</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation and Problem Statement</td>
<td>2</td>
</tr>
<tr>
<td>1.1.1 Missing Data Races</td>
<td>2</td>
</tr>
<tr>
<td>1.1.2 Lacking Practical Analyses</td>
<td>4</td>
</tr>
<tr>
<td>1.2 Providing Practical High-Coverage Sound Predictive Race Detection</td>
<td>5</td>
</tr>
<tr>
<td>1.3 Dissertation Presentation of Contributions</td>
<td>7</td>
</tr>
<tr>
<td>2. Background and Terminology</td>
<td>9</td>
</tr>
<tr>
<td>2.1 Execution Model</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Detecting Data Races</td>
<td>11</td>
</tr>
<tr>
<td>2.3 Predictive Relations and Analyses</td>
<td>13</td>
</tr>
<tr>
<td>2.3.1 Reordered Execution Trace</td>
<td>13</td>
</tr>
<tr>
<td>2.3.2 Causally-Precedes Relation and Analysis</td>
<td>16</td>
</tr>
<tr>
<td>2.3.3 Weak-Causally-Precedes Relation and Analysis</td>
<td>17</td>
</tr>
<tr>
<td>2.3.4 Limitations of Prior Work</td>
<td>20</td>
</tr>
<tr>
<td>2.4 Implementation</td>
<td>20</td>
</tr>
<tr>
<td>2.5 Methodology</td>
<td>22</td>
</tr>
</tbody>
</table>
### 3. Raptor: Online Set-Based Dynamic Analysis for Sound Predictive Race Detection

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Introduction</td>
<td>24</td>
</tr>
<tr>
<td>3.2 Terminology and Motivation</td>
<td>26</td>
</tr>
<tr>
<td>3.2.1 Execution Model Extended</td>
<td>26</td>
</tr>
<tr>
<td>3.2.2 Limitations of Recursive Ordering</td>
<td>28</td>
</tr>
<tr>
<td>3.3 Raptor Overview</td>
<td>30</td>
</tr>
<tr>
<td>3.4 Raptor’s Analysis State and Invariants</td>
<td>33</td>
</tr>
<tr>
<td>3.4.1 Program-Order Set: $PO(\rho)$</td>
<td>34</td>
</tr>
<tr>
<td>3.4.2 Happens-Before Set: $HB(\rho)$</td>
<td>35</td>
</tr>
<tr>
<td>3.4.3 Causally-Precedes Set: $CP(\rho)$</td>
<td>36</td>
</tr>
<tr>
<td>3.4.4 Conditionally Causally-Precedes Set: $CCP(\rho)$</td>
<td>37</td>
</tr>
<tr>
<td>3.5 The Raptor Analysis</td>
<td>39</td>
</tr>
<tr>
<td>3.5.1 Handling Write Events</td>
<td>40</td>
</tr>
<tr>
<td>3.5.2 Handling Read Events</td>
<td>42</td>
</tr>
<tr>
<td>3.5.3 Handling Acquire Events</td>
<td>43</td>
</tr>
<tr>
<td>3.5.4 Handling Release Events</td>
<td>44</td>
</tr>
<tr>
<td>3.5.5 Removing Obsolete Sets and Detecting CP-Races</td>
<td>47</td>
</tr>
<tr>
<td>3.5.6 Correctness of detecting CP-races</td>
<td>48</td>
</tr>
<tr>
<td>3.6 Examples</td>
<td>50</td>
</tr>
<tr>
<td>3.7 Correctness: Raptor Tracks CP Soundly and Completely</td>
<td>55</td>
</tr>
<tr>
<td>3.8 Evaluation</td>
<td>71</td>
</tr>
<tr>
<td>3.8.1 Implementation</td>
<td>72</td>
</tr>
<tr>
<td>3.8.2 Methodology</td>
<td>73</td>
</tr>
<tr>
<td>3.8.3 Coverage and Performance</td>
<td>74</td>
</tr>
<tr>
<td>3.9 Contribution and Impact</td>
<td>79</td>
</tr>
</tbody>
</table>

### 4. Doesn’t-Commute and Weak-Doesn’t-Commute Predictive Analysis

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Introduction</td>
<td>80</td>
</tr>
<tr>
<td>4.2 Doesn’t-Commute Relation</td>
<td>81</td>
</tr>
<tr>
<td>4.3 Completeness</td>
<td>84</td>
</tr>
<tr>
<td>4.4 Doesn’t-Commute Analysis</td>
<td>88</td>
</tr>
<tr>
<td>4.5 Weak-Doesn’t-Commute Relation</td>
<td>91</td>
</tr>
<tr>
<td>4.6 Weak-Doesn’t-Commute Analysis</td>
<td>92</td>
</tr>
<tr>
<td>4.7 Contribution and Impact</td>
<td>94</td>
</tr>
</tbody>
</table>

### 5. Vindicator: Providing Soundness for Unsound High-Coverage Predictive Race Detection

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.</td>
<td>96</td>
</tr>
</tbody>
</table>
5.1 Introduction ........................................ 96
5.2 Vindicating Predictable Races .......................... 97
  5.2.1 The Constraint Graph ............................ 99
  5.2.2 Adding Constraints to \( G \) ......................... 102
  5.2.3 Detecting Cycles in the Constraint Graph ........ 104
  5.2.4 Constructing a Reordered Trace ................. 106
  5.2.5 Asymptotic Complexity ......................... 109
5.3 Limitations of \textsc{VindicateRace} ...................... 109
  5.3.1 \textsc{ConstructReorderedTrace} ................. 109
  5.3.2 \textsc{AddConstraints} .......................... 112
5.4 Evaluation ........................................... 112
  5.4.1 Implementation .................................. 112
  5.4.2 Methodology ..................................... 115
  5.4.3 Detection Coverage of Predictable Races ........ 115
  5.4.4 Run-Time Performance ......................... 120
5.5 Contribution and Impact .............................. 123

6. SmartTrack: Optimizing Predictive Race Detection ................. 125
  6.1 Introduction ........................................ 125
  6.2 Limitations of Existing Race Detection Analyses ........ 126
  6.3 SmartTrack: Optimized Predictive Analysis ............ 130
    6.3.1 Epoch and Ownership Optimizations ............... 131
    6.3.2 Optimizing Detecting Conflicting Critical Sections 135
    6.3.3 Performance Cost of Soundness for DC Analysis .... 144
  6.4 Evaluation ........................................... 145
    6.4.1 Implementation .................................. 145
    6.4.2 Methodology ..................................... 146
    6.4.3 Run-Time Characteristics ....................... 147
    6.4.4 Comparing Baselines ............................ 150
    6.4.5 Run-Time and Memory Performance ............... 153
    6.4.6 Predictable Race Coverage ..................... 155
    6.4.7 Evaluation Summary ............................ 156
  6.5 Contribution and Impact .............................. 157

7. Related Work .......................................... 159

8. Conclusion ............................................ 163
  8.1 Contributions ....................................... 163
  8.2 Impact .............................................. 164
List of Tables

Table Page

3.1 Dynamic execution characteristics and detected HB- and CP-races reported for the evaluated programs. *Events* is the number of program events processed by FastTrack and Raptor, and *#Thr* is the execution’s thread count. *Time* is wall clock execution time, reported in seconds or hours and rounded to two significant figures or the nearest integer if $\geq 100$. For each race column (HB&CP, HB, CP), the first number is distinct static races, and the second number (in parentheses) is dynamic races. Each run of FastTrack, Raptor, and Datalog CP is limited to 2 hours; *N/A* columns indicate that Datalog CP did not finish in 2 hours and thus did not report any CP-races. 75

3.2 Event distances for detected CP-races. For each static CP-only race reported by Raptor, the table reports a range of event distances for all dynamic occurrences of the static CP-only race. 78

5.1 HB-, WCP-, and DC-races detected by our implementation. *DC-races* include all WCP-races; *WCP-races* include all HB-races. **VINDICATERACE confirmed that every DC-race is a true predictable race.** Each result is the average of 10 trials, rounded to the nearest integer. 116

5.2 Characteristics of the nine static DC-only races reported by Vindicator in our experiments. Each race is an unordered pair of static locations, which are represented as class, method, and line number. *Event distance* is the range of event distances (distance apart in $\llbracket t \rrbracket$ between the two conflicting events) across all dynamic instances of the race, from a separate experiment that totally orders events. 117

5.3 Characteristics of **VINDICATERACE** (Algorithm 10) for all dynamic DC-only races across the 10 trials. 120
5.4 Run time and memory usage for Vindicator and other analysis configurations. All values (except thread counts) are rounded to 2 significant figures. The table reports Vindicator’s slowdown relative to uninstrumented execution as the first value in the Vindicator column (e.g., 33 × for avrora). The text explains other values.

6.1 Evaluated analyses. Optimizations increase from left to right, and relations weaken from top to bottom.

6.2 Frequencies of non-same-epoch reads and writes for SmartTrack-WDC, for each evaluated program.

6.3 Run-time characteristics of the evaluated programs. “NSEAs” are non-same-epoch accesses.

6.4 Run-time and memory performance for FastTrack-based HB analyses and for unoptimized DC and WDC analyses, relative to uninstrumented execution.

6.5 Geometric mean of run time and memory usage across the evaluated programs.

6.6 Run time, relative to uninstrumented execution, of various analyses for each evaluated program.

6.7 Memory usage, relative to uninstrumented execution, of various analyses for each evaluated program.

6.8 Average races reported by various analyses for each evaluated program (excluding batik and lusearch, for which all analyses report no races). In each cell, the first value is statically distinct races (i.e., distinct program locations) and the second value, in parentheses, is total dynamic races. As the text explains, significant differences between the algorithms (Unopt-, FTO-, ST-) are attributable to how the algorithms behave after detecting the first race.
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A simple program in which x and y are shared variables, and this is a lock. The code in (a) represents a simple program. The trace in (b) represents a possible interleaving of the code that results in a data race, causing the program to crash. An HB analysis can detect the race, i.e., ( rd(x) \not\prec_{HB} wr(x) ). The trace in (c) has no data race, i.e. ( rd(x) \prec_{HB} wr(x) ).</td>
</tr>
<tr>
<td>2.2</td>
<td>Two executions in which x, y, and z are shared variables, and m is a lock. The single arrow represents a CP edge established by rule (a) of the CP definition (Definition 10). The differences between the executions are in bold.</td>
</tr>
<tr>
<td>2.3</td>
<td>The example execution in (a) has no HB-race (i.e., ( wr(x) \prec_{HB} rd(x) )), no CP-race (i.e., ( wr(x) \prec_{CP} rd(x) ), and a WCP-race (i.e., ( wr(x) \prec_{WCP} rd(x) )). The example execution in (a) has a predictable race, as the reordered execution in (b) demonstrates.</td>
</tr>
<tr>
<td>2.4</td>
<td>The example execution in (a) has no WCP-race (i.e., ( wr(x) \prec_{WCP} rd(x) )), but it has a predictable race, as the reordered execution in (b) demonstrates.</td>
</tr>
<tr>
<td>3.1</td>
<td>An example execution in which ( wr(x)^1 \prec_{CP} rd(x)^1 ). The last column shows, for each event ( e ), orderings relevant to ( wr(x)^1 \prec_{CP} rd(x)^1 ) for the subtrace up to and including ( e ). The arrow represents a CP ordering established by Rule (a) of the CP definition.</td>
</tr>
<tr>
<td>3.2</td>
<td>The invariants maintained by the Raptor analysis at every event in the observed total order.</td>
</tr>
</tbody>
</table>
3.3 Raptor’s analysis state updates for the execution from Figure 2.2(b). The cells under Analysis state changes show, after each event, the changes to each set owner’s sets (“—” indicates no changes). By default, the cells show additions to a set; the prefix “\” indicates removal from a set. For example, after rel(m)^2, Raptor adds T2 and m to CP(x^1) and removes T2:m^1 from CCP(x^1).

3.4 The execution from Figure 3.1, in which wr(x)^1 \prec_{cp} rd(x)^1 \_T^3, with Raptor’s analysis state updates shown in the same format as Figure 3.3. For brevity, this figure and the article’s remaining figures omit showing updates to PO(\rho). 52

3.5 An execution illustrating CCP transfer in which wr(x)^1 \prec_{cp} wr(x)^2, in the same format as Figure 3.3. For space, the figure omits set updates for x^2 and y^2.

3.6 A more complex execution than Figure 3.5 showing CCP transfer, where wr(x)^1 \prec_{cp} wr(x)^2. For space, the figure omits set updates for x^2 and y^2.

4.1 The execution in (a) has a predictable race and a DC-race (wr(x) \not\prec_{wdc} rd(x)) but no WCP-race (wr(x) \not\prec_{wcp} rd(x)). sync(o) is an abbreviation for the sequence acq(o); rd(oVar); wr(oVar); rel(o).

4.2 Depictions of two cases of the Lemma 1 proof.

4.3 Each execution has a DC-race, i.e., wr(x) \not\prec_{dc} rd(x), but no predictable race. The arrows show DC ordering (excluding PO ordering). sync(o) is an abbreviation for the sequence acq(o); rd(oVar); wr(oVar); rel(o).

4.4 An execution with a WDC-race that is not a predictable race. Solid arrows indicate WDC (also DC) ordering, and the dashed arrow indicates DC-only ordering. sync(o) represents the sequence acq(o); rd(oVar); wr(oVar); rel(o).

5.1 After adding consecutive-event constraint. Execution has a predictable race. Figure 2.4(b) (page 18) shows the reordered trace constructed by CONSTRUCTREORDEREDTRACE.

5.2 After adding consecutive-event constraints (left) and then LS constraints (right). The resulting constraint graph is cyclic, and the execution has no predictable race.
5.3 After adding consecutive-event and LS constraints. Execution has a predictable race. Figure 4.1(b) (page 83) shows the reordered trace constructed by CONSTRUCTREORDEREDTRACE.

5.4 After adding consecutive-event and LS constraints. The resulting constraint graph is cyclic, and the execution has no predictable race.

5.5 An execution with a predictable race and a DC-race ($wr(x) \not\ll_{dc} rd(x)$). Solid arrows are the initial DC constraints, and dashed arrows are constraints added by ADDCONSTRAINTS. The circled numbers (e.g., 1) represent the order of events prepended to a reordered trace by CONSTRUCTREORDEREDTRACE. A seventh event cannot be prepended to the trace without violating the graph constraints or lock semantics.

5.6 We use this figure to represent three different example executions: (a) an execution omitting the sync(s6) events, which has a predictable race; (b) an execution omitting the sync(s4) events, which has a predictable race; and (c) the unmodified execution, which has no predictable race. The arrows represent DC ordering.

5.7 Cumulative distribution of the event distance of three kinds of dynamic races. For a given event distance, the plot shows the percentage of dynamic races with at least that event distance.

6.1 Example executions used by the text to illustrate how SmartTrack-DC works. All arrows show DC ordering. sync(o) represents the sequence acq(o); rd(oVar); wr(oVar); rel(o).
Chapter 1: Introduction

The advance of modern parallel software systems provides developers with more cores but requires efficient use of parallel infrastructure to reach computing power capable of achieving the ever-growing demands for faster and more efficient applications. Parallel software is ubiquitous in modern times, and the current trend is to utilize more cores by exploiting parallelization across them to continually increase performance.

Modern shared-memory programming languages, including C++ and Java, achieve parallelization by providing semantics that allow compiler and processor optimizations under the language’s memory model. Memory models describe possible behaviors of a program, but only programs containing no data races guarantees that all executions will appear to be sequentially consistent or correct in an equivalent single-threaded execution.

A shared-memory program has a data race if two conflicting memory accesses (accesses, at least one of which is a write, to the same memory location by different threads) can execute consecutively (with no interleaving program operations). Modern shared-memory programming languages achieve the desired performance at the sacrifice of correctness by providing undefined or ill-defined semantics for executions with data races [2, 10–13, 58, 88]. Thus data races represent a rising threat to system reliability, leaving systems vulnerable to fatal crashes, data corruption, and other unexpected errors [9, 17, 19, 32, 44, 46, 52, 55, 68, 73, 83, 88, 94].
1.1 Motivation and Problem Statement

The challenge in writing and diagnosing shared-memory programs is that data races manifest nondeterministically under specific thread interleavings, program inputs, and execution environments such that they may stay hidden even for extensively tested programs [94]. Thus it is crucial to provide tools to expose data races that can manifest within a program execution. However, current techniques in data race detection are limited in data race coverage or remain impractical in run time and memory usage performance.

Problem Statement. To accurately write and diagnose efficient shared-memory programs requires robust dynamic analyses that expose all data races from the observed execution within run time acceptable for in-house testing. Existing predictive analyses fail to detect all knowable races from the observed execution and remain impractical compared to industry standard race detectors. The lack of a practical high-coverage sound data race detector leaves shared-memory programs vulnerable to data races and developers unable to reliably trust their application.

1.1.1 Missing Data Races

The prominent method of detecting and reporting races is dynamic analysis that is implemented by commercial detectors [43, 86, 87]. The prevailing dynamic analysis that detects data races is happens-before (HB) analysis, which detects conflicting accesses unordered by the HB relation [49].

While HB analysis is (like all dynamic analyses) limited to analyzing the observed execution, a weakness of HB analysis is that it misses some races that are knowable from the
observed execution. Recently, researchers have developed predictive analyses that compute a partial order that detects more races than HB analysis [20, 40, 41, 47, 54, 57, 72, 79, 80, 85, 92].

This section discusses the limited data race coverage of dynamic analysis and predictive analysis. A further discussion of other major categories of data race detection, such as static analysis, hybrid analysis, and other data race detection techniques can be found in Chapter 7.

**Dynamic analysis.** Dynamic analysis techniques detect conflicting accesses unordered by the happens-before relation, the union of program and synchronization order of events within an execution [26, 31, 49, 75]. Alternatively, other analyses detect similar races based on conflicting regions or forcing conflicting events to execute simultaneously [5, 6, 24, 28, 83, 95]. These techniques are sound; any detected data race can occur in a real execution. However, in addition to being limited to analyzing the observed execution, these techniques miss data races that are knowable, can definitely occur in some execution, from the observed execution.

Due to the nondeterminism in shared-memory programs, techniques that detect data races can require tens or hundreds of runs or more to manifest a race [104] and take weeks to reproduce, diagnose, and fix in production systems [36, 55]. Production-time analysis finds data races that occur in production settings, using sampling to trade coverage for performance [5, 14, 28, 45, 59, 89, 103] or requiring custom hardware support [22, 74, 82, 101, 104]. Unsurprisingly, data race detectors do not see much use in practice, both in testing and production environments [59].

**Predictive analysis.** Predictive dynamic analysis proves a promising direction for detecting data races by finding races within interleavings different from the observed execution
using either an SMT-based approach or a partial-order-based approach [20, 40, 41, 47, 54, 80, 85, 92].

SMT-based approaches encode reordering constraints as SMT constraints and employ an SMT solver to detect races [20, 40, 41, 54, 80, 85]. These approaches reason about the set of all feasible reorderings within a trace which is exponential and must be approximated, as through a windowing strategy, to achieve execution within reasonable run time and memory usage. These predictive techniques are unable to scale to full program executions, requiring the use of bounded windows of analysis, thus missing races that are “far apart” in the execution [20, 40, 41, 54, 80, 85, 92].

Alternatively, partial-order-based approaches analyze a program execution to track the order of events based on a partial relation over the program trace [47, 92]. The analysis uses the partial order to reason about reordered program traces while enforcing the same program behavior as the observed execution. While Smaragdakis et al.’s causally-precedes (CP) analysis requires bounded windows of the analysis to execute in a reasonable time, thus missing races that are “far apart.” Kini et al.’s weak-causally-precedes (WCP) analysis can analyze full program executions at the cost of an overly conservative partial relation, thus missing races.

Sound predictive analyses only detect true data races but miss reporting races that exist within a program execution.

1.1.2 Lacking Practical Analyses

This section discusses how high-coverage predictive analysis remains impractical in run time and memory usage performance.
Happens-before analysis sets the standard for practical dynamic data race detection—roughly $8 \times$ slowdown according to prior work [31, 35]. Most prior work in predictive analysis is unable to scale to full program executions, remaining impractical and missing races. State-of-the-art partial-order-based predictive analysis increases race detection capability but remains impractical compared with happens-before data race detectors. Achieving higher race coverage than happens-before analysis is accomplished by reducing the ordering between critical sections while ensuring the same behavior as the observed execution. Maintaining the necessary metadata to track ordering between critical sections results in poor performance for existing predictive analyses, a cost industry standard happens-before race detectors do not have.

Predictive techniques lack sufficient coverage in reasonable run time and memory usage for developers to reliably trust their application.

1.2 Providing Practical High-Coverage Sound Predictive Race Detection

We present a body of work that aims to increase race detection capability beyond existing predictive analyses and close the performance gap between predictive analysis and highly optimized HB analysis.

Increasing race coverage. We improve the race detection capability of predictive analysis by developing new partial orders and analyses that detect more races than prior work and develop a new algorithm for predictive analysis that can scale to far-apart races.

To begin, we strengthen race detection capability with new predictive partial orders that weaken the observed order of executing events more than state-of-the-art predictive partial orders—thus detecting more races than existing predictive analyses. While these new predictive analyses are unsound (report false races) we find that they rarely or never
report false races in practice. Furthermore, to strictly improve race coverage we ensure the same soundness guarantees as existing industry standard detectors by developing an algorithm that verifies if potential races detected by our new predictive analyses are true races, resulting in a sound approach overall.

We indirectly improve race detection capability with a new online analysis for a partial order with an inherent recursive nature. Our online analysis summarizes the recursive nature, finding races that the limited coverage of the bounded online analysis performed by prior work is unable to detect.

**Improving performance.** We improve the run time and memory usage performance by developing an algorithm that achieves practical predictive analysis comparable with highly optimized happens-before data race detectors. Our algorithm optimizes the cost of predictive analyses in two main ways: by being the first to apply similar epoch optimizations used by popular happens-before race detectors to predictive race detection directly and by applying similar epoch optimizations to maintain metadata for ordering critical sections. Our optimizations apply to a family of predictive analyses, thus achieving practical predictive analysis that competes with industry standard detectors.

**Contribution and Impact**

This dissertation introduces a set of techniques that improve every aspect of predictive analysis with the goal of achieving a practical high-coverage sound predictive analysis. The work described in this dissertation increases the data race coverage potential for predictive analysis. We demonstrate detection of data races that prior work is unable to detect with scalable analyses tracking sufficiently weak partial orders. This work also improves the
run time and memory usage performance of predictive analysis. We show that the developed optimizations applied to a family of predictive analyses improve performance to be competitive with industry standard detectors. A practical high-coverage sound predictive analysis is a promising alternative to HB analysis that provides developers a tool that can detect more races than prior work in competitive time and space to current widely used data race detectors.

The insights of this dissertation to advance the state-of-the-art in predictive analysis opens the opportunity for further weakening and redesign of existing predictive partial orders. We show the existence of and a need to detect more data races than prior work. Future work can further improve the data race coverage and performance of predictive analysis and other techniques that rely on dynamic analysis information about a program, allowing for higher race coverage or more efficient analysis techniques for data race detection. This work makes a case for predictive analysis to be the prevailing approach for detecting data races. This work and future development provide developers with practical use tools that guarantee reliable concurrent software. As a result, developers will be more inclined to regularly use predictive analysis during testing so that they can reliably write and diagnose shared-memory programs.

1.3 Dissertation Presentation of Contributions

This dissertation presents the following work divided into chapters:

- Background on data races, existing predictive analyses and corresponding relations, and closely related work is discussed in Chapter 2.

- Chapter 3 presents our work Raptor, a sound predictive analysis that computes the CP relation online soundly and completely. Raptor is concurrent work with WCP and
is the first work to tackle the challenging task of computing CP online soundly and completely.

- **Chapter 4** introduces two predictive analyses, doesn’t-commute and weak-doesn’t-commute, that track new partial orders doesn’t-commute relation and weak-doesn’t-commute relation, respectively.

- **Chapter 5** introduces *Vindicator* [79], our work on high-coverage sound predictive race detection. This chapter aims to provide soundness guarantees for both the doesn’t-commute and weak-doesn’t-commute analyses.

- **Chapter 6** presents our optimization algorithm for predictive analysis, *SmartTrack*, a practical solution to predictive race detection. This chapter aims to optimize predictive analyses to be competitive with highly optimized HB analysis.

- Related work is discussed in **Chapter 7**.

- The impact of this work is examined, and the dissertation is concluded in **Chapter 8**.
Chapter 2: Background and Terminology

This chapter formally defines the execution model and important notation used throughout the dissertation. We then overview dynamic analysis and how data races are detected. Lastly, we describe predictive analyses from prior work and explain their motivations and limitations.

2.1 Execution Model

An execution trace \( tr \) is a totally ordered list of events, denoted \(<_{tr}\), that represents a linearization of events in a multithreaded execution.\(^1\) Each event consists of a thread identifier (e.g., \( T1 \) or \( T2 \)) and an operation with the form \( wr(x) \), \( rd(x) \), \( acq(m) \), or \( rel(m) \), where \( x \) is a variable and \( m \) is a lock. Throughout the dissertation we often denote events simply by their operation (e.g., \( wr(x) \) or \( acq(m) \)) and use the helper function \( thr(e) \) to get the event’s associated thread.

We assume that any observed execution trace is \textit{well formed}, meaning a thread only acquires a lock that is not held and only releases a lock it holds, and lock release order is last in, first out (i.e., critical sections are well-nested).

Two events \( e \) and \( e' \) in \( tr \) are totally ordered, \( e <_{tr} e' \), if \( e \) occurs before \( e' \) in the observed order, and \( e \leq_{tr} e' \) if \( e <_{tr} e' \) \& \( e = e' \). The figures throughout this dissertation display

\(^1\) We can safely assume sequential consistency (SC) [48] because Java, C++, and other languages provide SC until the first data race [2, 11, 58].
example execution traces in top-to-bottom order, such as the traces in Figures 2.1(b) and 2.1(c), representing observed execution order ($<_{tr}$) and column placement representing the executing thread.

We define helper functions $A(r)$ that returns the acquire event starting the critical section ended by release event $r$, and $R(a)$ that returns the release event ending the critical section started by acquire event $a$. Function $CS(r)$ returns the set of events in the critical section ended by release event $r$, including $r$ and $A(r)$. That is, $CS(r) \equiv \{ e \mid thr(e) = thr(r) \land A(r) \leq_{tr} e \leq_{tr} r \}$. Events $e$ and $e'$ are conflicting, denoted $e \simeq e'$, if one is a write event and the other is a read or write event to the same variable and $thr(e) \neq thr(e')$.

**Definition 1** (Program-order). *Program-order* ($PO$) is a strict partial order, denoted $\prec_{po}$, that orders events executed by the same thread. For two events $e$ and $e'$, $e \prec_{po} e'$ if $e <_{tr} e' \land thr(e) = thr(e')$.

**Definition 2** (Data Race). A shared-memory program has a *data race* if an execution of the program has two memory accesses that are conflicting and consecutive. Two memory access events $e$ and $e'$ are:

- **conflicting** if both accesses are to the same memory location by different threads and at least one access is a write, i.e., $e \simeq e'$

- **consecutive** if there are no interleaving program operations, i.e., $e <_{tr} e' \land \exists e'' \mid e <_{tr} e'' <_{tr} e'$

Figure 2.1(a) is a simple program that contains a data race. The program counts the number of times $y$ calls $compute(\ldots)$ and clears $y$ before a new count starts. The execution trace in Figure 2.1(b) represents a possible observed execution trace of the program. The
Initially \( x = 1; y = \text{new} \ldots \);

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{synchronized(this)} ) { ( \text{if (x == 0)} ) ( y = \text{null}; ) ( y = \text{new} \ldots ; ) ( y.\text{compute(\ldots)} ; ) ( x++ ; ) } )</td>
<td>( \text{synchronized(this)} ) { ( \text{y = null; } ) ( \text{y = new} \ldots ; ) ( \text{y.\text{compute(\ldots)};} ) ( x = 0; ) }</td>
</tr>
</tbody>
</table>

\[
\text{(a) Simple Program}
\]

<table>
<thead>
<tr>
<th>( \text{Thread 1} )</th>
<th>( \text{Thread 2} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{acq(this)} )</td>
<td>( \text{wr(y)} )</td>
</tr>
<tr>
<td>( \text{wr(x)} )</td>
<td>( \text{rd(x)} )</td>
</tr>
<tr>
<td>( \text{rd(x)} )</td>
<td>( \text{rd(y)} )</td>
</tr>
</tbody>
</table>

\[
\text{(b) Example Execution with a data race}
\]

<table>
<thead>
<tr>
<th>( \text{Thread 1} )</th>
<th>( \text{Thread 2} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{acq(this)} )</td>
<td>( \text{wr(y)} )</td>
</tr>
<tr>
<td>( \text{rel(this)} )</td>
<td>( \text{wr(x)} )</td>
</tr>
</tbody>
</table>

\[
\text{(c) Example Execution with no data race}
\]

![Figure 2.1: A simple program in which \( x \) and \( y \) are shared variables, and this is a lock. The code in (a) represents a simple program. The trace in (b) represents a possible interleaving of the code that results in a data race, causing the program to crash. An HB analysis can detect the race, i.e., \( \text{rd(x)} \not\prec_{\text{HB}} \text{wr(x)} \). The trace in (c) has no data race, i.e. \( \text{rd(x)} \prec_{\text{HB}} \text{wr(x)} \).}

events \( \text{wr(x)} \) by Thread 2 and \( \text{rd(x)} \) by Thread 1 form a data race which leads to postponing the assignment of \( x \) by Thread 2 until after reading the value of \( x \) by Thread 1, resulting in a program crash since \( y \) dereferences a null.

Figure 2.1(b) is not the execution interleaving intended by the developer. To detect and expose the data race we focus on dynamic program analysis.

### 2.2 Detecting Data Races

Dynamic program analysis detects data races by observing executing program memory accesses and synchronization operations (i.e., events) during program execution. Dynamic analyses are *sound* if they report only true races, no false races are reported. Soundness is an essential property because each reported race—whether true or false—takes substantial time for developers to investigate [4, 31, 36, 59, 68].
The most common sound dynamic analysis for detecting data races is the *happens-before* (HB) *analysis*, which detects conflicting accesses unordered by the HB relation, a strict partial order that orders events by the union of PO and synchronization order [26, 31, 49, 75, 77].

**Definition 3** (Happens-before). Given a trace $tr$, $\prec_{\text{HB}}$ is the smallest relation such that:

- Two events are ordered by HB if they are ordered by PO. That is, $e \prec_{\text{HB}} e'$ if $e \prec_{\text{PO}} e'$.

- Release and acquire events on the same lock are ordered by HB (a.k.a. synchronization order). That is, $r \prec_{\text{HB}} a$ if $r$ and $a$ are release and acquire events, respectively, on the same lock and $r \prec_{tr} a$.

- HB is transitively closed. That is, $e \prec_{\text{HB}} e'$ if $\exists e'' | e \prec_{\text{HB}} e'' \land e'' \prec_{\text{HB}} e'$.

Happens-before approaches track the HB relation [49] to determine the order between events during an execution. If the HB analysis does not order two conflicting accesses, then an execution exists in which the two conflicting events will execute consecutively, indicating a data race. So HB approaches provide sound (i.e., no false positive) data race detection within the current execution, but cannot detect races in other executions that change the synchronization order from the observed execution.

Consider the example program execution in Figure 2.1(b), in which the HB relation orders the accesses to $y$ (because the HB relation orders the critical sections on this). The accesses to $x$ are unordered by the HB relation, and therefore a data race is detected on $x$ by HB analysis.

However, the HB relation cannot detect a data race in a different execution. The trace in Figure 2.1(c) shows the HB relation ordering $\text{rel}(\text{this})$ by Thread 1 and $\text{acq}(\text{this})$ by Thread 2, establishing the HB order $\text{rd}(x) \prec_{\text{HB}} \text{wr}(x)$ and missing the race on $x$. 

12
This example highlights the run-to-run nondeterminism of data races and the limitation of HB analysis. If the trace in Figure 2.1(b) rarely executes, then the HB analysis may need to test a program tens of thousands to millions of times before an execution interleaving matches the trace. If a data race exists, then a feasible reordering must also exist that exposes the race. We focus on predictive analyses tracking partial orders that detect more races than the HB analysis from the observed execution.

2.3 Predictive Relations and Analyses

*Predictive analyses* detect data races that are possible in executions *other than* the observed execution. *Sound* predictive analyses detect more races than HB without reporting false races [20, 40, 41, 47, 54, 80, 85, 92]. For the rest of this chapter, we focus on sound partial-order-based predictive analyses that weaken the observed order of events more than the HB relation to detect otherwise hidden races.

2.3.1 Reordered Execution Trace

Sound predictive race detection analyzes a trace \( tr \) and detects data races that occur in some other, unobserved execution. We define a *correctly reordered* trace \( tr' \) as any trace that must be feasible because the observed trace \( tr \) executed:

**Definition 4** (Correct reordering). A trace \( tr' \) is a correct reordering of \( tr \) if \( tr' \) contains only events in \( tr \) and the following properties hold:

*Program-order (PO) rule:* Two PO-ordered events must execute in the same order in \( tr' \) as in \( tr \) unless the second event is not in \( tr' \). That is, \( e \prec_{PO} e' \implies (e \prec_{tr'} e' \vee e' \notin tr') \).
Conflicting accesses (CA) rule: Two conflicting events must execute in the same order in $tr'$ as in $tr$ unless the second event is not in $tr'$. That is, $(e <_{tr} e' \land e \simeq e') \implies (e <_{tr'} e' \lor e' \notin tr')$.

Lock semantics (LS) rule: Critical sections on the same lock cannot overlap. That is, if $a_1$ and $a_2$ are acquire events on the same lock, $a_1 <_{tr'} a_2 \implies R(a_1) <_{tr'} a_2$.

Note that the PO and CA rules constrain reordering based on ordering in $tr$, while the LS rule constrains $tr'$ directly.

Note that the CA rule, which prohibits reordering all pairs of conflicting accesses, is overly strict: it disallows some reordered traces in which every read has the same last write as in $tr$. For example, given $rd(x) <_{tr} wr(x)$, the CA rule disallows a $tr'$ that contains $wr(x)$ but not $rd(x)$, although such a reordered trace is not necessarily invalid. However, we do not know how to encode a less-restrictive CA rule in a partial order (such as prior work’s relations [47, 92]), e.g., how to encode that $tr'$ can reorder a read–write conflict only if the read is not in the reordered execution.

**Definition 5** (Predictable race). An execution trace $tr$ has a predictable race if it has two conflicting events $e_1$ and $e_2$ ($e_1 \simeq e_2$) that in some correctly reordered trace $tr'$ are consecutive ($e_1 <_{tr'} e_2 \land \exists e \mid e_1 <_{tr'} e <_{tr'} e_2$).

**Definition 6** (Soundness). A relation, analysis, or approach is sound if it detects no race for every execution trace $tr$ that has no predictable race.

**Definition 7** (Completeness). A relation, analysis, or approach is complete if it finds a race for every execution trace $tr$ that has a predictable race.

Note that completeness means detecting all predictable races knowable from an observed execution trace, not all of a program’s data races. These definitions of soundness and
completeness, which follow the predictive data race detection literature (e.g., [40, 47, 92]), are swapped compared with most work on static and dynamic (non-predictive) race detection (cf. Chapter 7).

We consider the prior work discussed in this dissertation as predictive because detected unordered conflicting accesses indicate a predictable race or a race that occurs in a correct reordering of the observed execution. Throughout this dissertation, we consider HB data race detection non-predictive contrary to other literature in predictive analysis [47]. However, we consider an execution trace to have an HB-race if it has two conflicting events that are unordered by HB. Thus we consider predictive relations able to detect more than just HB-races, which helps distinguish HB analysis from the other analyses (WCP, DC, and WDC analyses) that this dissertation introduces and/or optimizes.

HB misses predictable races as shown in Figure 2.2 with an execution that has a predictable race, but HB orders the conflicting events to \( x \), i.e., \( \text{wr}(x) \prec_{\text{HB}} \text{rd}(x) \). The HB

Figure 2.2: Two executions in which \( x \), \( y \), and \( z \) are shared variables, and \( m \) is a lock. The single arrow represents a CP edge established by rule (a) of the CP definition (Definition 10). The differences between the executions are in bold.
relation’s race coverage limitation is due to ordering all critical sections of the same lock regardless of the shared variable accesses protected by them.

2.3.2 Causally-Precedes Relation and Analysis

Smaragdakis et al. introduce the causally-precedes (CP) relation [92], a strict partial order that is strictly weaker than HB that conservatively orders conflicting accesses that may not race in some other execution. CP is a sound relation because all conflicting accesses not ordered by CP are definitely HB races in some other execution. The following presentation of the CP relation is similar to prior work’s [92].

Definition 8 (Causally-precedes). Given a trace $tr$, $\prec_{CP}$ is the smallest relation such that:

(a) Release and acquire operations on the same lock containing conflicting events are CP ordered. That is, $\text{rel}(m) \prec_{CP} \text{acq}(m)$ if $\exists e \exists e' | e <_{tr} e' \land e \simeq e' \land A(\text{rel}(m)) \prec_{PO} e \prec_{PO} \text{rel}(m) \land \text{acq}(m) \prec_{PO} e' \prec_{PO} R(\text{acq}(m))$.

(b) Two critical sections on the same lock are CP ordered if they contain CP-ordered events. Because of the next rule, this rule can be expressed simply as follows: $\text{rel}(m) \prec_{CP} \text{acq}(m)$ if $A(\text{rel}(m)) \prec_{CP} R(\text{acq}(m))$.

(c) CP is closed under left and right composition with HB. That is, $e \prec_{CP} e'$ if $\exists e'' | e \prec_{HB} e'' \prec_{CP} e'$ or if $\exists e'' | e \prec_{CP} e'' \prec_{HB} e'$.

An execution trace $tr$ has a CP-race if it has two if it has two events $e <_{tr} e'$ such that $e \simeq e' \land e \not\prec_{PO} e' \land e \not\prec_{CP} e'$.

More precisely, two conflicting accesses unordered by CP imply either a data race or a deadlock in another execution [92].
The CP relation detects more races than the HB relation as seen in the execution trace in Figure 2.2(a). The HB relation orders \( \text{wr}(x) \prec_{HB} \text{rd}(x) \) because \( \text{wr}(x) \prec_{PO} \text{rel}(m) \prec_{HB} \text{acq}(m) \prec_{PO} \text{rd}(x) \). In contrast, the CP relation captures the fact that the critical sections do not have conflicting accesses, so \( \text{rel}(m) \not\prec_{CP} \text{acq}(m) \) and \( \text{wr}(x) \not\prec_{CP} \text{rd}(x) \), thus detecting the predictable race on \( x \). However, in Figure 2.2(b), \( \text{wr}(x) \prec_{CP} \text{rd}(x) \) because CP captures the fact that a data race may not occur if the critical sections execute in the reverse order. So \( \text{rel}(m) \prec_{CP} \text{acq}(m) \) by CP Rule (a), and it follows that \( \text{wr}(x) \prec_{CP} \text{rd}(x) \) by CP Rule (c) because \( \text{wr}(x) \prec_{HB} \text{rel}(m) \prec_{CP} \text{acq}(m) \prec_{HB} \text{rd}(x) \).

Smaragdakis et al. show how to compute CP in polynomial time in the execution length. However, their analysis cannot scale to full executions, and instead analyzes bounded execution windows of 500 consecutive events [92], missing CP-races (conflicting accesses unordered by CP) involving accesses that are “far apart” in the observed execution. Most existing predictive analyses cannot scale beyond analyzing bounded windows of execution, thus missing predictable races whose accesses are “far apart” in an observed execution.

CP is incomplete: it misses predictable races within a bounded window of consecutive events. Figure 2.3(a) shows an execution with a predictable race, but CP orders the conflicting events to \( x \), i.e., \( \text{wr}(x) \prec_{CP} \text{rd}(x) \). The reordered execution in Figure 2.3(b) shows a valid execution that exposes the data race between the \( \text{wr}(x) \) and \( \text{rd}(x) \). The CP relation is incomplete because of the overly conservative ordering of critical sections and its composition with synchronization order.

### 2.3.3 Weak-Causally-Precedes Relation and Analysis

Kini et al. recently introduced the weak-causally-precedes (WCP) relation [47], a strict partial order that is weaker than CP and thus predicts more races than CP (WCP is also
Figure 2.3: The example execution in (a) has no HB-race (i.e., \( wr(x) \prec_{HB} rd(x) \)), no CP-race (i.e., \( wr(x) \prec_{CP} rd(x) \)), and a WCP-race (i.e., \( wr(x) \prec_{WCP} rd(x) \)). The example execution in (a) has a predictable race, as the reordered execution in (b) demonstrates.

Figure 2.4: The example execution in (a) has no WCP-race (i.e., \( wr(x) \prec_{WCP} rd(x) \)), but it has a predictable race, as the reordered execution in (b) demonstrates.
weaker than HB and thus predicts more races than HB.) The following presentation of the WCP relation is similar to prior work’s [47].

**Definition 9** (Weak-causally-precedes). Given a trace \( tr \), \( \prec_{WCP} \) is the smallest relation such that:

(a) If two critical sections on the same lock contain conflicting events, then the first critical section is ordered by WCP to the second conflicting event. That is, \( r_1 \prec_{WCP} e_2 \) if \( r_1 \) and \( r_2 \) are release events on the same lock, \( r_1 \prec_tr\ r_2 \), \( e_1 \in CS(r_1) \), \( e_2 \in CS(r_2) \), and \( e_1 \preceq e_2 \).

(b) Release events on the same lock are ordered by WCP if their critical sections contain WCP-ordered events. Because of the next rule, this rule can be expressed simply as follows: \( r_1 \prec_{WCP} r_2 \) if \( r_1 \) and \( r_2 \) are release events on the same lock and \( A(r_1) \prec_{WCP} r_2 \).

(c) WCP is closed under left and right composition with HB. That is, \( e \prec_{WCP} e' \) if \( \exists e'' \mid e \prec_{HB} e'' \prec_{WCP} e' \) or \( e \prec_{WCP} e'' \prec_{HB} e' \).

An execution trace has a **WCP-race** if it has two conflicting events that are unordered by the strict partial order \( \prec_{WCP} \cup \prec_{PO} \). The example execution in Figure 2.3(a) has a WCP-race on the two conflicting events to \( x \). The WCP relation makes a distinction between the order of events within a critical section, resulting in the \( rel(m) \) by Thread 1 to be WCP-ordered to \( rd(z) \), thus \( wr(x) \not\prec_{WCP} rd(x) \).

WCP is the weakest known predictive relation that is also sound. Furthermore, WCP can be computed with a dynamic analysis that scales to whole execution traces [47]. However,

---

3Two conflicting accesses unordered by WCP imply either a data race or a deadlock in another execution [47].
WCP is incomplete. Figure 2.4(a) shows an execution that has no WCP-race but has a predictable race, as Figure 2.4(b) demonstrates the predictable race. WCP’s incompleteness is due to its composition with synchronization order. Furthermore, WCP can be computed with a dynamic analysis in linear time with respect to the size of the program trace but slows programs by tens of times compared with the native execution of the program. WCP remains impractical, leaving a significant performance gap with highly optimized HB analysis.

2.3.4 Limitations of Prior Work

In summary, industry standard race detectors based on the happens-before analysis detect data races within reasonable run time and memory usage but miss races that manifest within executions knowable from the observed execution. Prior work in predictive analyses that can detect more races than happens-before analysis have poor performance and still miss races that are knowable from the observed execution.

As discussed throughout this chapter, pioneering predictive analysis causally-precedes limits scalability by detecting races within bounded windows and still miss races that are knowable from the observed execution. The weak-causally-precedes analysis detects more races than any existing predictive analysis but still misses races that are knowable from the observed execution and remains impractical compared to industry standard happens-before detectors. SMT-based predictive analysis and alternative approaches are unable to scale to full executions and miss races that are discussed in more detail in Section 7.

2.4 Implementation

The implementations evaluated in Chapters 3, 5, and 6 of this dissertation are built on RoadRunner, a dynamic analysis framework for concurrent Java programs [33] that

\[\text{https://github.com/stephenfreund/RoadRunner}\]
implements the high-performance FastTrack happens-before race detector [31, 34, 35, 77]. RoadRunner provides analysis hooks at memory accesses and synchronization operations by instrumenting Java bytecode dynamically at class loading time. RoadRunner generates events for memory accesses (loads and stores to field and array elements) and synchronization operations (lock acquire, release, wait, and resume; thread fork and join; and volatile read and write). The evaluation in Chapter 3 uses RoadRunner version 0.3 where the evaluation in Chapters 5 and 6 use RoadRunner version 0.5, leading to differences in methodology.

Handling events. Each analysis discussed throughout this dissertation handle read, write, acquire, and release events as described in the algorithms for each analysis. Additionally, each analysis supports additional synchronization primitives. Each analysis establishes order on thread fork and join; between conflicting volatile variable accesses; and from “class initialized” to “class accessed” events. Each analysis treats wait() as a release followed by an acquire.

Handling races. In theory, each analysis discussed in this dissertation handles executions up to the first race. In practice, similar to industry-standard race detectors [43, 86, 87], our analysis implementations continue analyzing executions after the first race. At a race, an analysis first reports the race, including the static program location that detected the race, in order to report more races to users and collect results for full executions.

The differences in algorithms between the chapters lead to variation in how races are handled. Chapter 3 uses reference counting for each access independently to identify and

5Alternatively, both static program locations for racing accesses could be reported, similar to Google’s ThreadSanitizer [86, 87], but at the cost of tracking additional information.
report all races detected. In contrast, in Chapters 5 and 6, if an analysis detects multiple races at an access (e.g., a write races with multiple last readers) we count it as a single race to avoid the complexity of determining whether some of the detected races are dependent on each other.

Then each analysis in Chapters 3 and 5 perform best-effort ordering between the accesses involved in the race, updating last-access metadata to essentially make the execution race free, avoiding future false races that are precluded by the earlier detected race. After the analyses in Chapter 6 detects a race, it continues normally without best-effort ordering.

2.5 Methodology

Each analysis is evaluated using the DaCapo benchmarks [7], version 9.12-bach, which are real programs harnessed as benchmarks. For the analysis in Chapter 3, we use small workload sizes and exclude programs that do not run out of the box with RoadRunner due to the provided support of RoadRunner version 0.3. For the analyses in Chapters 5 and 6, we use RoadRunner version 0.5’s provided support for harnessing and running the DaCapo programs (e.g., for dynamic bytecode instrumentation in the presence of DaCapo’s custom class loading); the provided workloads are close to DaCapo’s default workload size. RoadRunner does not currently support eclipse, tradebeans, or tradesoap; our evaluation excludes those programs, as well as fop since it is single threaded.

The experiments in Chapter 3 execute on a quiet system with four Intel Xeon E5-4620 8-core processors (32 cores total) with hyperthreading disabled and 128 GB of main memory, running Linux 2.6.32. The experiments in Chapters 5 and 6 run on a quiet system with an Intel Xeon E5-2683 14-core processor with hyperthreading disabled and 256 GB of main memory running Linux 3.10.0. We configure RoadRunner to tell programs that there are
8 available cores; as a result, several DaCapo programs create 8 worker threads. We run RoadRunner with the HotSpot 1.8.0 JVM and let it choose and adjust the heap size on the fly.
Chapter 3: Raptor: Online Set-Based Dynamic Analysis for Sound Predictive Race Detection

3.1 Introduction

Predictive analyses continuously target improving race detection capability but still miss races within the observed execution. Weakening observed ordering requires sound predictive analyses to either perform costly reasoning about reordered thread interleavings or conservatively ordering events that preserve a valid reordered execution. The costly reasoning or conservative ordering restricts many predictive analyses from scaling to entire execution traces for real, large programs [20, 40, 41, 47, 54, 80, 85, 92]. While pioneering partial-order-based predictive analysis causally-precedes’ race coverage is limited to bounded windows of the observed execution, it remains impractical because the causally-precedes analysis is an offline analysis. The contributions of this chapter develop an online analysis that tracks the causally-precedes relation, thus increasing race coverage and providing a predictive analysis that scales to long executions.

The only partial-order-based predictive analysis, at the time of developing the contributions of this chapter, causally-precedes (CP) analysis by Smaragdakis et al. illustrates how to compute the CP relation in polynomial time in the execution length. However, the causally-precedes analysis cannot scale to full program executions due to the recursive
nature of the CP relation. Instead, the analysis requires analyzing bounded windows of the program trace to execute within reasonable run time and memory usage. Reducing full program traces to bounded windows of 500 consecutive events [92] causes causally-precedes analysis to miss data races involving access that are “far apart” (accesses separated by more events than the bounded window size) in the observed execution.

Predictive analysis must scale to full program executions to build a practical high-coverage sound predictive analysis. This chapter addresses the issue of scalability by introducing a novel dynamic analysis Raptor (Race predictor) that computes the CP relation online soundly and completely that scales to long executions. Raptor provides a scalable solution to tracking the CP relation online by capturing the dependent recursive nature of the CP relation during analysis and avoids the need to “look back” at the entire program execution trace as a whole. During the time of development, we found Raptor was the first predictive analysis to scale to long executions, being able to detect races that all prior work was unable to detect.

Raptor is inherently an online analysis because it summarizes an execution’s behavior so far in the form of analysis state, rather than needing to look at the entire execution so far. Raptor’s key insights lie in how it captures the dependent, recursive nature of the CP relation. Raptor maintains analysis state for the HB and CP relations and for “conditional CP”—potential CP relations that hold if some other CP relation holds. We introduce analysis invariants and prove that Raptor maintains the invariants after each step of a program’s execution, meaning that it soundly and completely tracks CP.

We have implemented Raptor as a dynamic analysis for Java programs. Our unoptimized prototype implementation can analyze executions of real programs with hundreds of thousands or millions of events within an hour or two. In contrast, Smaragdakis et al.’s analysis
generally cannot scale beyond bounded windows of thousands of events [92]. As a result, Raptor detects some CP-races that are too “far apart” for the offline CP analysis to detect.

While concurrent work’s weak-causally-precedes (WCP) analysis by Kini et al. [47] is faster and (as a result of using a weaker relation than CP) detects more races than Raptor, computing CP online is a challenging problem that prior work has been unable to solve [47, 92]. Furthermore, Raptor provides the first set-based algorithm for partial-order-based predictive analysis. Though recent advances in predictive analysis have subsumed Raptor’s race coverage and performance, the alternative technique of a set-based approach provides unique avenues for future development.

Raptor advances the state of the art by (1) being the first online analysis for computing CP soundly and completely and (2) demonstrably scaling to long executions and finding real CP-races that the prior CP analysis cannot detect.

3.2 Terminology and Motivation

This section extends the execution model introduced in Chapter 2 and motivates the challenges of computing the causally-precedes (CP) relation online.

3.2.1 Execution Model Extended

The Raptor analysis (discussed in Section 3.5) tracks distinct events, such as differentiating the first write to a shared variable from the second write to the same shared variable, at any given point in the execution. We extend the execution model, introduced in Chapter 2 (page 9), with additional information per event to identify distinct events.

An event is one of \( wr(x)^i \), \( rd(x)^i_T \), \( acq(m)^i \), or \( rel(m)^i \), where \( x \) is a variable, \( m \) is a lock, and \( i \) specifies the \( i \)th instance of the event, i.e., \( wr(x)^i \) is the \( i \)th write to variable \( x \). \( rd(x)^i_T \) is a read by thread \( T \) to variable \( x \) such that \( wr(x)^i <_{tr} rd(x)^i_T <_{tr} wr(x)^{i+1} \).
The following presentation extends the CP relation, introduced in Chapter 2 (Definition 8), to use the extended execution model.

**Definition 10 (Causally-precedes).** Given a trace \( tr \), \( \prec_{cp} \) is the smallest relation such that:

(a) Release and acquire operations on the same lock containing conflicting events are CP ordered. That is, \( \text{rel}(m)^i \prec_{cp} \text{acq}(m)^j \) if \( \exists e \exists e' | e <_{tr} e' \land e \simeq e' \land \text{acq}(m)^j \prec_{po} e \prec_{po} \text{rel}(m)^i \).

(b) Two critical sections on the same lock are CP ordered if they contain CP-ordered events. Because of the next rule, this rule can be expressed simply as follows: \( \text{rel}(m)^i \prec_{cp} \text{acq}(m)^j \) if \( \text{acq}(m)^j \prec_{cp} \text{rel}(m)^i \).

(c) CP is closed under left and right composition with HB. That is, \( e \prec_{cp} e' \) if \( \exists e'' | e \prec_{hb} e'' \prec_{cp} e' \) or if \( \exists e'' | e \prec_{cp} e'' \prec_{hb} e' \).

For simplicity, the rest of this chapter refers to the rules of the CP definition (page 27) as Rules (a), (b), and (c). Throughout this chapter, an execution trace \( tr \) consists of events observed in a total order, denoted \( \leq_{tr} \) (reflexive) or \( <_{tr} \) (irreflexive). Furthermore, we make use of reflexive variants of \( \prec_{po} \) and \( \prec_{hb} \), defined as \( \leq_{po} \) and \( \leq_{hb} \), respectively, when it is correct to do so.

**Example.** Consider the execution in Figure 3.1 (page 28), ignoring the rightmost column (explained in Section 3.2.2). The accesses to \( x \) are CP ordered through the following logic: \( \text{rel}(u)^1 \prec_{cp} \text{acq}(u)^2 \) by Rule (a) implies \( \text{acq}(m)^1 \prec_{cp} \text{rel}(m)^2 \) by Rule (c), which implies \( \text{rel}(m)^1 \prec_{cp} \text{acq}(m)^2 \) by Rule (b). Since \( \text{wr}(x)^1 \prec_{hb} \text{rel}(m)^1 \prec_{cp} \text{acq}(m)^2 \prec_{hb} \text{rd}(x)^{1}_{T3} \), \( \text{wr}(x)^1 \prec_{cp} \text{rd}(x)^{1}_{T3} \) by Rule (c).
### Figure 3.1: An example execution in which $\text{wr}(x)^1 \prec_{\text{CP}} \text{rd}(x)^1_{T3}$. The last column shows, for each event $e$, orderings relevant to $\text{wr}(x)^1 \prec_{\text{CP}} \text{rd}(x)^1_{T3}$ for the subtrace up to and including $e$. The arrow represents a CP ordering established by Rule (a) of the CP definition.

#### 3.2.2 Limitations of Recursive Ordering

This chapter targets the challenge of developing an online analysis for tracking the CP relation and detecting CP-races. An online analysis must (1) compute CP soundly and completely; (2) maintain analysis state that summarizes the execution so far, without needing to maintain and refer to the entire execution trace; and (3) analyze real program execution traces using time and space that is acceptable for heavyweight in-house testing.

The main difficulty in tracking the CP relation online is in summarizing the execution so far as analysis state. An analysis can compute the PO and HB relations for events executed so far based only on the events executed so far. In contrast, an online CP analysis must handle the fact that CP may order two events because of later events. For example, $e \prec_{\text{CP}} e'$. 

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>Relevant orderings “knowable” after event</th>
</tr>
</thead>
<tbody>
<tr>
<td>wr(x)$^1$</td>
<td></td>
<td></td>
<td>wr(x)$^1 \prec_{\text{HB}} \text{rel}(m)^1$ [knowable at acq(m)$^1$ since rel(m)$^1$ is inevitable]</td>
</tr>
<tr>
<td>acq(m)$^1$</td>
<td></td>
<td></td>
<td>acq(m)$^1 \prec_{\text{HB}} \text{rel}(u)^1$</td>
</tr>
<tr>
<td>acq(u)$^1$</td>
<td></td>
<td></td>
<td>acq(u)$^1 \prec_{\text{PO}} \text{wr}(y)^1 \prec_{\text{PO}} \text{rel}(u)^1$</td>
</tr>
<tr>
<td>acq(v)$^1$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rel(v)$^1$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rd(x)$^1_{T3}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>acq(m)$^2$</td>
<td></td>
<td></td>
<td>acq(m)$^2 \prec_{\text{HB}} \text{rd}(x)^1_{T3}$</td>
</tr>
<tr>
<td>acq(u)$^2$</td>
<td></td>
<td></td>
<td>acq(u)$^2 \prec_{\text{HB}} \text{rel}(m)^2$</td>
</tr>
<tr>
<td>acq(v)$^2$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rel(v)$^2$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rd(y)$^1_{T2}$</td>
<td></td>
<td></td>
<td>rd(y)$^1_{T2} \prec_{\text{PO}} \text{rel}(y)^1_{T2} \prec_{\text{PO}} \text{rel}(u)^2$. rel(u)$^1 \prec_{\text{CP}} \text{acq}(u)^2$.</td>
</tr>
<tr>
<td>rel(u)$^2$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rel(m)$^2$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
only because of a future event \( e'' \) (\( e' <_{tr} e'' \)); we provide a concrete example shortly. The analysis must summarize the possible order between \( e \) and \( e' \) at least until \( e'' \) executes, without needing access to the entire execution trace. Smaragdakis et al. explain the inherent challenge of developing an online analysis for CP as follows [92]:

"CP reasoning, based on [the definition of CP], is highly recursive. Notably, Rule (c) can feed into Rule (b), which can feed back into Rule (c). As a result, we have not implemented CP using techniques such as vector clocks, nor have we yet discovered a full CP implementation that only does online reasoning (i.e., never needs to “look back” in the execution trace)."

Smaragdakis et al.’s CP algorithm encodes the recursive definition of CP in Datalog, guaranteeing polynomial-time execution in the size of the execution trace. However, the algorithm is inherently offline because it fundamentally needs to “look back” at the entire execution trace. Experimentally, Smaragdakis et al. find that their algorithm does not scale to full program traces. Instead, they limit their algorithm’s computation to bounded windows of 500 consecutive events [92].

Example. Figure 3.1 illustrates the challenge of developing an online analysis that computes CP soundly and completely while handling the recursive nature of the CP definition. The last column shows the orderings relevant to \( \text{wr}(x)^1 \prec_{\text{cp}} \text{rd}(x)^1_{T3} \) that are “knowable” after each event \( e \). More formally, these are orderings that exist for a subtrace comprised of events up to and including \( e \).

The accesses to \( x \) are CP ordered through the following logic: \( \text{rel}(u)^1 \prec_{\text{cp}} \text{acq}(u)^2 \) by Rule (a) implies \( \text{acq}(m)^1 \prec_{\text{cp}} \text{rel}(m)^2 \) by Rule (c), which implies \( \text{rel}(m)^1 \prec_{\text{cp}} \text{acq}(m)^2 \) by Rule (b). Since \( \text{wr}(x)^1 \prec_{\text{HB}} \text{rel}(m)^1 \prec_{\text{cp}} \text{acq}(m)^2 \prec_{\text{HB}} \text{rd}(x)^1_{T3}, \text{wr}(x)^1 \prec_{\text{cp}} \text{rd}(x)^1_{T3} \) by Rule (c).

6Other existing predictive analyses are either unsound (reporting false races), or (like the existing CP work [92]) are limited to analyzing bounded windows of execution [20, 40, 41, 47, 54, 80, 85].
The \( \text{wr}(x)^1 \prec_{CP} \text{rd}(x)^1_{T3} \) because \( \text{rel}(m)^1 \prec_{CP} \text{acq}(m)^2 \). However, at \( \text{rd}(x)^1_{T3} \), an online analysis cannot determine that \( \text{rel}(m)^1 \prec_{CP} \text{acq}(m)^2 \) (and thus \( \text{wr}(x)^1 \prec_{CP} \text{rd}(x)^1_{T3} \)) based on the execution subtrace so far. Not until \( \text{rd}(y)^1_{T2} \) is it knowable that \( \text{rel}(u)^1 \prec_{CP} \text{acq}(u)^2 \) and thus \( \text{acq}(m)^1 \prec_{CP} \text{rel}(m)^2 \), \( \text{rel}(m)^1 \prec_{CP} \text{acq}(m)^2 \), and \( \text{wr}(x)^1 \prec_{CP} \text{rd}(x)^1_{T3} \). A sound and complete online analysis for CP must track analysis state that captures ordering once it is knowable without maintaining the entire execution trace.

Alternatively, consider instead that \( T1 \) executed the critical section on \( u \) before the critical section on \( m \). In that subtly different execution, \( \text{wr}(x)^1 \not\prec_{CP} \text{rd}(x)^1_{T3} \). A sound and complete online analysis for CP must track analysis state that captures the difference between these two execution variants.

We note that more challenging examples exist. For instance, it is possible to modify the example so that it is unknowable even at \( \text{rel}(m)^2 \) that \( \text{acq}(m)^1 \prec_{CP} \text{rel}(m)^2 \). Section 3.5 presents three such examples.

### 3.3 Raptor Overview

*Raptor* (Race predictor) is an online dynamic analysis that computes the CP relation soundly and completely by maintaining analysis state that captures CP orderings knowable for a subtrace of events up to and including the latest event in the execution. This section overviews the components of Raptor’s analysis state.

**Terminology.** Throughout the rest of this chapter, we say that an event \( e \) is *CP ordered to* a lock \( m \) or thread \( T \) if there exists an event \( e' \) that releases \( m \) (\( \exists i \mid e' = \text{rel}(m)^i \)) or is executed by \( T \) (\( \text{thr}(e') = T \)), respectively, and \( e \prec_{CP} e' \). This property, in turn, implies that for any future event \( e'' \) (i.e., \( e' <_{ir} e'' \)), \( e \prec_{CP} e'' \) if \( e'' \) acquires \( m \) (\( e'' = \text{acq}(m)^j \)) or is executed by
\( T \left( thr(e''') = T \right) \), respectively, since CP composes with HB. Similarly, \( e \) is HB ordered to a lock or thread if the same conditions hold for \( \prec_{HB} \) instead of \( \prec_{CP} \).

**Sets.** Existing HB analyses typically represent analysis state using vector clocks [31,64,75]. Since the CP relation conditionally orders critical sections, conditional information is required on synchronization objects to accurately track CP. Using sets to track the HB and CP relations in terms of synchronization objects naturally manages conditional information, compared with using vector clocks. Raptor’s analysis state is represented by sets containing synchronization objects—locks and threads—that represent CP, HB, and PO orderings. For example, if a lock \( m \) is an element of the HB set \( HB(x^8) \), it means that the 8th write of \( x \) event, \( wr(x)^8 \), is HB ordered to \( m \). Similarly, the thread element \( T2 \in CP(y^3_{T1}) \) means that the event \( rd(y)^3_{T1} \) (a read by \( T1 \) to \( y \) between the 3rd and 4th writes to \( y \)) is CP ordered to thread \( T2 \). Raptor’s sets are most related to the sets used by Goldilocks, a sound and complete HB data race detector [26].

**Sets for each access to a variable.** As implied above, rather than each variable \( x \) having CP, HB, and PO sets, every access \( wr(x)^i \) and \( rd(x)^i_T \) has its own CP, HB, and PO sets. Per-access CP sets are necessary because of the nature of the CP relation: at \( wr(x)^{i+1} \), it is not in general knowable whether \( wr(x)^i \prec_{CP} wr(x)^{i+1} \) or \( \forall_T rd(x)^i_T \prec_{CP} wr(x)^{i+1} \). Similarly, it is not generally knowable at \( rd(x)^i_T \) whether \( wr(x)^i \prec_{CP} rd(x)^i_T \). In Figure 3.1, even after \( rd(x)^1_{T3} \) executes, Raptor must continue to maintain sets for \( wr(x)^1 \) because \( wr(x)^1 \prec_{CP} rd(x)^1_{T3} \) has not yet been established.

Maintaining per-access sets would seem to require massive time and space (proportional to the length of the execution), making it effectively an offline analysis like prior work’s CP analysis [92]. However, as we show in Section 3.5.5, Raptor can safely remove sets under
detectable conditions, e.g., it can remove \( \text{wr}(x)^i \)'s sets once it determines that \( \text{wr}(x)^i \prec \text{cp} \text{wr}(x)^{i+1} \) and \( \forall T \text{ wr}(x)^i \prec \text{cp} \text{ rd}(x)^i_T \).

Sets for lock acquires. Raptor tracks CP, HB, and PO sets not just for variable accesses, but also for lock acquire operations to compute CP order by Rule (b) (i.e., \( \text{acq}(m)^i \prec \text{cp} \text{ rel}(m)^j \) implies \( \text{rel}(m)^i \prec \text{cp} \text{ acq}(m)^j \)). For example, \( T3 \in CP(m^i) \) means the event \( \text{acq}(m)^i \) is CP ordered to thread \( T3 \).

Similar to sets for variable accesses, maintaining a CP, HB, and PO set for each lock acquire might consume high time and space proportional to the execution’s length. In Section 3.5.5, we show how Raptor can safely remove an acquire \( \text{acq}(m)^i \)'s sets once they are no longer needed—once no other CP ordering is dependent on the possibility of \( \text{acq}(m)^i \) being CP ordered with a future \( \text{rel}(m) \).

Conditional CP sets. As mentioned earlier, it is unknowable in general at an event \( e' \) whether \( e \prec \text{cp} e' \). This recursive nature of the CP definition prevents immediate determination of CP ordering at \( e' \). This delayed knowledge is unavoidable due to Rule (b), which states that \( \text{rel}(m)^j \prec \text{cp} \text{ acq}(m)^j \) if \( \text{acq}(m)^i \prec \text{cp} \text{ rel}(m)^j \). A CP ordering might not be known until \( \text{rel}(m)^j \) executes—or even longer because Rule (c) can “feed into” Rule (b), which can feed back into Rule (c) [92].

Raptor maintains conditional CP (CCP) sets to track the fact that, at a given event in an execution, CP ordering may or may not exist, depending on whether some other CP ordering exists. For example, an element \( n:m^j \) (or \( T:m^j \)) in the CCP set \( \text{CCP}(x^i) \) means that \( \text{wr}(x)^i \) is CP ordered to lock \( n \) (or thread \( T \)) if \( \text{acq}(m)^j \prec \text{cp} \text{ rel}(m)^k \) for some future event \( \text{rel}(m)^k \).

In contrast with the above, Goldilocks does not need or use sets for each variable access, sets for lock acquires, or conditional sets, since it maintains sets that track only the HB.
relation [26]. The $HB$ components of Raptor are similar to $Goldilocks$ analysis [26]. However, the CCP sets unique to Raptor incur significant algorithmic complexity compared with $Goldilocks$. $Goldilocks$ and Raptor use per-variable “locksets” or sets, respectively, to track the HB relation (and in Raptor’s case, the CP relation) soundly and completely [26].

**Outline of Raptor presentation.** Section 3.4 describes Raptor’s sets and their elements in detail, and it presents invariants maintained by Raptor’s sets at every event in an execution trace. Section 3.5 introduces the Raptor analysis that adds and, in some cases, removes set elements at each execution event. Section 3.5.5 describes how Raptor removes “obsolete” sets and detects CP-races.

### 3.4 Raptor’s Analysis State and Invariants

This section describes the analysis state that Raptor maintains. Every set owner $\rho$, which can be a variable write instance $x_i$, a variable read instance $x_i^T$, or lock acquire instance $m_i$, has the following sets: $PO(\rho)$, $HB(\rho)$, $CP(\rho)$, and $CCP(\rho)$. In general, elements of each set are threads $T$ and locks $m$, with a few caveats: $HB(\rho)$ maintains an index for each lock element (e.g., $m_i$), and each $CCP(\rho)$ element includes an associated lock instance upon which CP ordering is conditional (e.g., $m:n^j$ or $T:n^j$). In addition, each set for a variable write instance $x_i$ and read instance $x_i^T$ can contain a special element $\xi$, which indicates ordering between $wr(x)^i$ and $wr(x)^{i+1}$ and between $rd(x)^i_T$ and $wr(x)^{i+1}$. Similarly, each set for $x_i^T$ can also contain a special element $\xi_T$ for each thread $T$, which indicates ordering between $wr(x)^i$ and $rd(x)^i_T$. Since knowledge of CP ordering may be delayed, a write or read instance could establish CP order to a thread $T$ at an event later than the conflicting write or read instance. The special elements are necessary to distinguish CP ordering to the conflicting write or read instance from CP ordering to a later event.
Let \( e \) be any event in the program trace. The following invariants hold for the point in the trace immediately before \( e \).

Let \( e_\xi = \text{wr}(x)^{h+1} \) if \( e = \text{wr}(x)^{h} \) or \( e = \text{rd}(x)^{h} \). Let \( e_{\xi_T} = \text{rd}(x)^{h} \) if \( e = \text{wr}(x)^{h} \). Otherwise \( (e \) is a lock acquire/release event), \( e_\xi \) and \( e_{\xi_T} \) are “invalid events” that match no real event.

We define a boolean function \( \text{appl}(\sigma, e') \) that evaluates to true iff event \( e' \) “applies to” set element \( \sigma \):

\[
\text{appl}(\sigma, e') := \begin{cases} 
\text{thr}(e') = T & \text{if } \sigma \text{ is a thread } T \\
\exists i \mid e' = \text{rel}(m)^i & \text{if } \sigma \text{ is a lock } m \\
e' = e_\xi & \text{if } \sigma = e_\xi \\
e' = e_{\xi_T} & \text{otherwise (} \sigma \text{ is } e_\xi\) 
\end{cases}
\]

The following invariants hold for every set owner \( \rho \). For each set owner \( \rho \), let \( e_\rho \) be the event corresponding to \( \rho \), i.e., \( e_\rho = \text{wr}(x)^{h} \) if \( \rho = x^{h} \), \( e_\rho = \text{rd}(x)^{h} \) if \( \rho = x^{h}_{T} \), or \( e_\rho = \text{acq}(m)^{h} \) if \( \rho = m^{h} \).

[PO] \( PO(\rho) = \{ \sigma \mid \sigma \text{ is not a lock } \land (\exists e' \mid \text{appl}(\sigma, e') \land e_\rho \prec_{\text{PO}} e' \prec_{\text{IR}} e) \} \)

[HB] \( HB(\rho) = \{ \sigma \mid (\exists e' \mid \text{appl}(\sigma, e') \land e_\rho \prec_{\text{HB}} e' \prec_{\text{IR}} e) \} \)

[HB-index] \( m^i \in HB(\rho) \iff (e_\rho \not\prec_{\text{HB}} \text{rel}(m)^{i-1} \land e_\rho \prec_{\text{HB}} \text{rel}(m)^{i} \prec_{\text{IR}} e) \)

[HB-critical-section] \( m^i \in HB(\rho) \iff (\text{acq}(m)^{i} \not\prec_{\text{PO}} e_\rho \prec_{\text{PO}} \text{rel}(m)^{i} \land (\rho = x^{h} \lor \rho = x^{h}_{T}) \land e_\rho \prec_{\text{IR}} e) \)

[CP] \( CP(\rho) \cup \{ \sigma \mid (\exists n^j \mid \sigma : n^k \in CCP(\rho) \land \exists j \mid \text{rel}(n)^{j} \prec_{\text{CP}} \text{acq}(n)^{j} \prec_{\text{IR}} e) \} = \{ \sigma \mid (\exists e' \mid \text{appl}(\sigma, e') \land e_\rho \prec_{\text{CP}} e' \prec_{\text{IR}} e) \}
\]

[CP-rule-A] \( (\exists m, T, h, i, e', e'' \mid \rho = m^{h} \land e_\rho \prec_{\text{PO}} e' \prec_{\text{PO}} \text{rel}(m)^{h} \prec_{\text{IR}} \text{acq}(m)^{j} \prec_{\text{PO}} e'' \prec_{\text{PO}} \text{rel}(m)^{i} \land e'' \prec_{\text{IR}} e \land \text{thr}(e'') = T \land e' \neq e'') \implies T \in CP(\rho) \)

[CCP-constraint] \( \exists \sigma : n^k \in CCP(\rho) \implies (\exists j \mid \text{acq}(n)^{j} \prec_{\text{IR}} e \land \text{rel}(n)^{j} \not\prec_{\text{IR}} e) \)

Figure 3.2: The invariants maintained by the Raptor analysis at every event in the observed total order.

Figure 3.2 shows invariants that the Raptor analysis maintains for every set owner \( \rho \).

The rest of this section explains these invariants in detail, using events \( e, e_\xi, e_{\xi_T}, \) and \( e_\rho \) as defined in the figure.

### 3.4.1 Program-Order Set: PO(\( \rho \))

According to the [PO] invariant in Figure 3.2, \( PO(\rho) \) contains all threads that the event \( e_\rho \) is PO ordered to. That said, we know from the definition of PO that \( e_\rho \) will be PO ordered to only one thread (the thread that executed \( e_\rho \)). In addition, for any
\( \rho = x^h \) or \( \rho = x^h_T \), \( PO(\rho) \) may contain the special element \( \xi \), indicating that \( wr(x)^h \) or \( rd(x)^h_T \), respectively, is PO ordered to the next write access to \( x \) by the same thread, i.e., \( wr(x)^h \prec_{po} wr(x)^{h+1} \) or \( rd(x)^h_T \prec_{po} wr(x)^{h+1} \). Similarly, for any \( \rho = x^h \), \( PO(\rho) \) may contain the special element \( \xi_T \), indicating that \( wr(x)^h \) is PO ordered to the next read access to \( x \) by thread \( T \), i.e., \( wr(x)^h \prec_{po} rd(x)^h_T \). Note that Raptor does not really need \( \xi \) and \( \xi_T \) to indicate \( wr(x)^h \prec_{hb} wr(x)^{h+1} \), \( rd(x)^h_T \prec_{hb} wr(x)^{h+1} \), or \( wr(x)^h \prec_{po} rd(x)^h_T \), since PO order is knowable at the next (read/write) access, but Raptor uses these elements for consistency with the CP and CCP sets, which do need \( \xi \) and \( \xi_T \) as explained later in this section.

### 3.4.2 Happens-Before Set: \( HB(\rho) \)

The \( HB(\rho) \) set contains threads and locks that the event \( e_\rho \) is HB ordered to. Figure 3.2 states three invariants for \( HB(\rho) \): the \([HB] \), \([HB-index]\), and \([HB-critical-section]\) invariants.

The \([HB]\) invariant defines which threads and locks are in \( HB(\rho) \). If \( e_\rho \) is HB ordered to a thread or lock, then \( HB(\rho) \) contains that thread or lock. This property implies that \( e_\rho \) will be HB ordered to any future event that executes on the same thread or acquires the same lock, respectively. Similar to PO sets, for \( \rho = x^h \) or \( \rho = x^h_T \), \( \xi \in HB(\rho) \) means \( wr(x)^h \prec_{hb} wr(x)^{h+1} \) or \( rd(x)^h_T \prec_{hb} wr(x)^{h+1} \), respectively. Additionally, for \( \rho = x^h \), \( \xi_T \in HB(\rho) \) means \( wr(x)^h \prec_{hb} rd(x)^h_T \). Though the \( \xi \) and \( \xi_T \) elements are superfluous (HB ordering is knowable at the next (read/write) access), Raptor maintains these elements for consistency with the CP and CCP sets that need it.

According to the \([HB-index]\) invariant, every lock \( m \) in \( HB(\rho) \) has a superscript \( i \) (e.g., \( m^i \)) that specifies the earliest release of \( m \) that \( e_\rho \) is HB ordered to. For example, \( m^i \in HB(\rho) \) means that \( e_\rho \prec_{hb} rel(m)^i \) but \( e_\rho \not\preceq_{hb} rel(m)^{i-1} \). This property tracks which
instance of the critical section on lock \( m, m' \), would need to be CP ordered to \( \text{rel}(m)^i \) to imply that \( e_\rho \prec_{\text{CP}} \text{acq}(m)^i \) (by Rules (b) and (c)).

According to the [HB-critical-section] invariant, for read/write accesses (\( \rho = x^h \) or \( \rho = x^h_T \)) only, \( m^i \) in \( HB(\rho) \) may have a subscript * (i.e., \( m^i_* \)), indicating that, in addition to \( e_\rho \prec_{\text{HB}} \text{rel}(m)^i \), \( e_\rho \) executed inside the critical section on lock \( m^i \), i.e., \( \text{acq}(m)^i \prec_{\text{PO}} e_\rho \prec_{\text{PO}} \text{rel}(m)^i \). Notationally, whenever \( m^i_* \in HB(\rho) \), \( m^i \in HB(\rho) \) is also implied. Raptor tracks this property to establish Rule (a) precisely.

### 3.4.3 Causally-Precedes Set: \( CP(\rho) \)

Analogous to \( HB(\rho) \) for HB ordering, each \( CP(\rho) \) set contains locks and threads that the event \( e_\rho \) is CP ordered to. However, at an event \( e' \), since Rule (b) may delay establishing \( e_\rho \prec_{\text{CP}} e' \), \( CP(\rho) \) does not necessarily contain \( \sigma \) such that \( \text{appl}(\sigma, e') \land e_\rho \prec_{\text{CP}} e' \). This property of CP presents two main challenges. First, Rule (b) may delay establishing CP order that is dependent on other CP orders. Raptor introduces the \( CCP(\rho) \) set (described below) to track potential CP ordering that may be established later. Raptor tracks every lock and thread that \( e_\rho \) is CP ordered to, either eagerly using \( CP(\rho) \) or lazily using \( CCP(\rho) \), according to the [CP invariant] in Figure 3.2.

Second, as a result of computing CP lazily, Raptor may not be able to determine that there is a CP-race between conflicting events \( \text{wr}(x)^i \preceq \text{wr}(x)^{i+1} \), \( \text{wr}(x)^i \preceq \text{rd}(x)^i_T \), or \( \text{rd}(x)^i_T \preceq \text{wr}(x)^{i+1} \) until after the second conflicting access event \( \text{rd}(x)^i_T \) or \( \text{wr}(x)^{i+1} \). For example, if the analysis adds \( T \) to \( CP(x^i) \) sometime after \( T \) executed \( \text{wr}(x)^{i+1} \), that does not necessarily mean that \( \text{wr}(x)^i \prec_{\text{CP}} \text{wr}(x)^{i+1} \) (it means only that \( \text{wr}(x)^i \) is CP ordered to some event by \( T \) after \( \text{wr}(x)^{i+1} \)). Raptor uses the special thread-like element \( \xi \) that represents the thread \( T \) up to event \( \text{wr}(x)^{i+1} \) only, so \( \xi \in CP(x^i) \) or \( \xi \in CP(x^i_T) \) only if \( \text{wr}(x)^i \prec_{\text{CP}} \text{wr}(x)^{i+1} \).
or \( \text{rd}(x)_T^i \prec_{\text{cp}} \text{wr}(x)^{i+1} \), respectively. Raptor also uses the special thread-like element \( \xi_T \) that represents the thread \( T \) up to event \( \text{rd}(x)_T^i \) only, so \( \xi_T \in \text{CP}(x^i) \) only if \( \text{wr}(x)^i \prec_{\text{cp}} \text{rd}(x)_T^i \).

The [\text{CP-rule-A} invariant] (Figure 3.2) covers the case for which Raptor always computes \( \text{CP} \) eagerly: when two critical sections on the same lock have conflicting events, according to Rule (a). In this case, the invariant states that if two critical sections, \( m^h \) and \( m^i \), are \( \text{CP} \) ordered by Rule (a) alone, then \( T \in \text{CP}(m^h) \) as soon as the second conflicting access executes. The [\text{CP-rule-A} invariant] is useful in proving that Raptor maintains the [\text{CP}] invariant (Section 3.7).

### 3.4.4 Conditionally Causally-Precedes Set: \( \text{CCP}(\rho) \)

Section 3.3 overviewed \( \text{CCP} \) sets. In general, \( \sigma : n^k \in \text{CCP}(\rho) \) means that the event \( e_\rho \) is \( \text{CP} \) ordered to \( \sigma \) if \( \text{acq}(n)^j \prec_{\text{cp}} \text{rel}(n)^j \), where \( n^j \) is an ongoing critical section (i.e., \( \text{acq}(n)^j <_{\text{tr}} e \land \text{rel}(n)^j \not<_{\text{tr}} e \)).

As mentioned above, the [\text{CP}] invariant says that for every \( \text{CP} \) ordering, Raptor captures it eagerly in a \( \text{CP} \) set or lazily in a \( \text{CCP} \) set (or both). A further constraint, codified in the [\text{CCP-constraint} invariant], is that \( \sigma : n^k \in \text{CCP}(\rho) \) only if a critical section on lock \( n \) is ongoing. As Section 3.5 shows, when \( n \)'s current critical section ends (at \( \text{rel}(n)^j \)), Raptor either (1) determines whether \( \text{acq}(n)^k \prec_{\text{cp}} \text{rel}(n)^j \) or (2) identifies another lock \( q \) that has an ongoing critical section such that it is correct to add some \( \sigma : q^f \) to \( \text{CCP}(\rho) \).

Like \( \text{CP}(\rho) \), when \( \rho = x^i \) or \( \rho = x^i_T \), \( \text{CCP}(\rho) \) can contain special thread-like elements of the form \( \xi : n^k \). The element \( \xi : n^k \in \text{CCP}(x^i) \) or \( \xi : n^k \in \text{CCP}(x^i_T) \) means that \( \text{wr}(x)^i \prec_{\text{cp}} \text{wr}(x)^{i+1} \) if \( \text{acq}(n)^k \prec_{\text{cp}} \text{rel}(n)^j \), or \( \text{rd}(x)_T^i \prec_{\text{cp}} \text{wr}(x)^{i+1} \) if \( \text{acq}(n)^k \prec_{\text{cp}} \text{rel}(n)^j \), respectively, where \( n^j \) is the current ongoing critical section of \( n \). Similarly, for \( \rho = x^i \), \( \text{CCP}(\rho) \) can contain special thread-like elements of the form \( \xi_T : n^k \). The element \( \xi_T : n^k \in \text{CCP}(x^i) \)
<table>
<thead>
<tr>
<th>Execution</th>
<th>Analysis state changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>x¹</td>
</tr>
<tr>
<td>T2</td>
<td>x²</td>
</tr>
<tr>
<td>wr(x)¹</td>
<td>PO HB T1</td>
</tr>
<tr>
<td>acq(m)¹</td>
<td>—</td>
</tr>
<tr>
<td>wr(y)¹</td>
<td>—</td>
</tr>
<tr>
<td>rel(m)¹</td>
<td>HB T1 m¹</td>
</tr>
<tr>
<td>rd(y)¹</td>
<td>CP T2 m¹</td>
</tr>
<tr>
<td>rel(m)²</td>
<td>CP T2 m¹</td>
</tr>
<tr>
<td>rd(x)²</td>
<td>CP T2 m¹</td>
</tr>
<tr>
<td></td>
<td>T2 T2:m¹</td>
</tr>
<tr>
<td></td>
<td>T2 T2:m¹</td>
</tr>
<tr>
<td></td>
<td>T2 T2:m¹</td>
</tr>
<tr>
<td></td>
<td>T2 T2:m¹</td>
</tr>
<tr>
<td></td>
<td>T2 T2:m¹</td>
</tr>
</tbody>
</table>

Figure 3.3: Raptor’s analysis state updates for the execution from Figure 2.2(b). The cells under Analysis state changes show, after each event, the changes to each set owner’s sets (“—” indicates no changes). By default, the cells show additions to a set; the prefix “\” indicates removal from a set. For example, after rel(m)², Raptor adds T2 and m to CP(x¹) and removes T2: m¹ from CCP(x¹).

means that wr(x)ⁱ ≺_{CP} rd(x)ᵀ if acq(n)ᵏ ≺_{CP} rel(n)ⁱ where nⁱ is the current ongoing critical section of n.

Raptor state example. Figure 3.3 shows updates to Raptor’s analysis state at each event for the execution from Figure 2.2(b). Directly after wr(y)¹, T1 ∈ PO(y¹), satisfying the [PO] invariant; and {T1,m¹} ∈ HB(y¹), satisfying the [HB], [HB-index], and [HB-critical-section] invariants since acq(m)¹ ≺_{PO} wr(y)¹ ≺_{PO} rel(m)¹. Directly after acq(m)², CCP(x¹), CCP(m¹), and CCP(y¹) contain T2 : m¹ satisfying the [CP] and [CCP-constraint] invariants, capturing the fact that T1’s events are CP ordered to T2 if acq(m)¹ ≺_{CP} rel(m)². Directly after rd(y)¹, T2 ∈ CP(m¹) satisfies the [CP-rule-A] invariant, and ξₜ₂:
$m^1 \in CCP(y^1)$ satisfies the [CP] invariant. Finally, after $\text{rel}(m)^2$ and $\text{rd}(x)^1_{T_2}$, $T_2 \in CP(x^1)$ and $\xi_{T_2} \in CP(x^1)$, respectively, satisfying the [CP] invariant, indicating $\text{wr}(x)^1 \prec_{cp} \text{rd}(x)^1_{T_2}$.

### 3.5 The Raptor Analysis

This section details Raptor, our novel dynamic analysis that maintains the invariants shown in Figure 3.2 and explained in Section 3.4. For each event $e$ in the observed trace $tr$, Raptor updates its analysis state by adding and (in some cases) removing elements from each set owner $\rho$’s sets. Assuming that the analysis state satisfies the invariants immediately before $e$, then at event $e$, Raptor modifies the analysis state so that it satisfies the invariants immediately after $e$:

**Theorem 1.** After every event, Raptor maintains the invariants in Figure 3.2.

Section 3.7 proves the theorem.

To represent the new analysis state immediately after $e$, we use the notation $PO(\rho)^+$, $HB(\rho)^+$, $CP(\rho)^+$, and $CCP(\rho)^+$. Initially, before Raptor starts updating analysis state at $e$ (i.e., immediately before $e$), $PO(\rho)^+ := PO(\rho)$, $HB(\rho)^+ := HB(\rho)$, $CP(\rho)^+ := CP(\rho)$, and $CCP(\rho)^+ := CCP(\rho)$.

**Initial analysis state.** Before the first event in $tr$, each $\rho$’s sets are initially empty, i.e., $PO(\rho) = HB(\rho) = CP(\rho) = CCP(\rho) = \emptyset$. This initial state conforms to Figure 3.2’s invariants for the point in execution before the first event.

To simplify checking for CP-races, the analysis assumes, for every program variable $x$, a “fake” initial access $\text{wr}(x)^0$. The analysis initializes $PO(x^0)$ to $\{\xi\} \cup \{\xi_T \mid T$ is a thread}; all other sets of $x^0$ are initially $\emptyset$. This initial state ensures that the first write access to $x$, $\text{wr}(x)^1$, appears to be ordered to the prior write access to $x$ ($\text{wr}(x)^0$), and any read accesses
to \( x \) before the first write (\( \text{rd}(x)^0_T \)) appear to be ordered to the prior write (\( \text{wr}(x)^0 \)), without requiring special logic to handle this corner case. (The analysis does not need fake accesses \( \text{rd}(x)^0_T \) because the analysis at the first write \( \text{wr}(x)^1 \) only checks for ordering with prior reads that actually executed.)

### 3.5.1 Handling Write Events

At a write event to shared variable \( x \), i.e., \( e = \text{wr}(x)^i_T \) by thread \( T \), the analysis performs the actions in Algorithm 1. The analysis establishes CP Rule (a); checks for PO, HB, CP, and conditional CP (CCP) order with prior accesses \( \text{wr}(x)^{i-1} \) and \( \text{rd}(x)^{i-1}_t \) for all threads \( t \); and initializes \( x^i \)'s sets.

### Algorithm 1: Raptor’s analysis for \( \text{wr}(x)^i_T \) by \( T \)

1. \( \textbf{for all } m \in \text{heldBy}(T) \) \textbf{do}
2. \( \quad \text{if } \exists j \exists h \mid m^j_t \in \text{HB}(x^h) \land T \notin \text{PO}(x^h) \text{ then } \text{CP}(m^j) \leftarrow \text{CP}(m^j) \cup \{T\} \) s.t. \( j \) is max satisfying index
3. \( \quad \textbf{for all threads } t \textbf{ do}
4. \( \quad \quad \text{if } \exists j \exists h \mid m^j_t \in \text{HB}(x^h) \land T \notin \text{PO}(x^h) \text{ then } \text{CP}(m^j) \leftarrow \text{CP}(m^j) \cup \{T\} \) s.t. \( j \) is max satisfying index
5. \( \quad \quad \quad \triangleright \text{Add } \xi \text{ to represent } T \text{ at } \text{wr}(x)^i \text{ from prior write } \text{wr}(x)^{i-1} \)
6. \( \quad \quad \text{if } T \in \text{PO}(x^{i-1}) \text{ then } \text{PO}(x^{i-1}) \leftarrow \text{PO}(x^{i-1}) \cup \{\xi\} \)
7. \( \quad \quad \text{if } T \in \text{HB}(x^{i-1}) \text{ then } \text{HB}(x^{i-1}) \leftarrow \text{HB}(x^{i-1}) \cup \{\xi\} \)
8. \( \quad \quad \text{if } T \in \text{CP}(x^{i-1}) \text{ then } \text{CP}(x^{i-1}) \leftarrow \text{CP}(x^{i-1}) \cup \{\xi\} \)
9. \( \quad \quad \textbf{for all threads } t \textbf{ do}
10. \( \quad \quad \quad \text{if } T \in \text{PO}(x^{i-1}) \text{ then } \text{PO}(x^{i-1}) \leftarrow \text{PO}(x^{i-1}) \cup \{\xi\} \)
11. \( \quad \quad \quad \text{if } T \in \text{HB}(x^{i-1}) \text{ then } \text{HB}(x^{i-1}) \leftarrow \text{HB}(x^{i-1}) \cup \{\xi\} \)
12. \( \quad \quad \quad \text{if } T \in \text{CP}(x^{i-1}) \text{ then } \text{CP}(x^{i-1}) \leftarrow \text{CP}(x^{i-1}) \cup \{\xi\} \)
13. \( \quad \quad \quad \textbf{for all } m \mid T : m \in \text{CCP}(x^{i-1}) \textbf{ do } \text{CCP}(x^{i-1}) \leftarrow \text{CCP}(x^{i-1}) \cup \{\xi : m\} \)
14. \( \quad \quad \quad \triangleright \text{Add } \xi \text{ to represent } T \text{ at } \text{wr}(x)^i \text{ from prior reads } \text{rd}(x)^{i-1}_t \text{ by all threads } t \)
15. \( \quad \quad \text{if } T \in \text{PO}(x^{i-1}) \text{ then } \text{PO}(x^{i-1}) \leftarrow \text{PO}(x^{i-1}) \cup \{\xi\} \)
16. \( \quad \quad \text{if } T \in \text{HB}(x^{i-1}) \text{ then } \text{HB}(x^{i-1}) \leftarrow \text{HB}(x^{i-1}) \cup \{\xi\} \)
17. \( \quad \quad \text{if } T \in \text{CP}(x^{i-1}) \text{ then } \text{CP}(x^{i-1}) \leftarrow \text{CP}(x^{i-1}) \cup \{\xi\} \)
18. \( \quad \quad \text{for all } m \mid T : m \in \text{CCP}(x^{i-1}) \textbf{ do } \text{CCP}(x^{i-1}) \leftarrow \text{CCP}(x^{i-1}) \cup \{\xi : m\} \)
19. \( \quad \quad \triangleright \text{Initialize sets for } x^i \)
20. \( \quad \text{PO}(x^i) \leftarrow \{T\} \)
21. \( \quad \text{HB}(x^i) \leftarrow \{T\} \cup \{m^j_t \mid m^j_t \in \text{heldBy}(T)\} \)
Establishing Rule (a). Lines 1–4 of Algorithm 1 show how the analysis establishes Rule (a) (conflicting critical sections are CP ordered). The helper function heldBy(T) returns the set of locks currently held by thread T (i.e., locks with active critical sections executed by T). For each lock m held by T, the analysis checks whether a prior conflicting access to x executed in an earlier critical section on m. Raptor adds T to CP(m^j) if a prior critical section m^j has executed a conflicting access, establishing Rule (a), which satisfies the [CP-rule-A] invariant (Figure 3.2). When T later releases m, the analysis will update each CP(ρ) set that depends on rel(m)^j ≺_{cr} acq(m)^k, as Section 3.5.4 describes.

Checking ordering with prior access. Lines 5–8 of Algorithm 1 add ξ to each set of x^{i-1} that already contains T, indicating PO, HB, and/or CP ordering from wr(x)^{i-1} to wr(x)^i, satisfying the invariants, e.g., if wr(x)^{i-1} ≺_{HB} wr(x)^i, then ξ ∈ HB(x^{i-1}) (part of the [HB] invariant). Notably, for any m^j such that T : m^j ∈ CCP(x^{i-1}), wr(x)^{i-1} ≺_{cr} wr(x)^i if acq(m)^j ≺_{cr} rel(m)^k (where m^k is the current critical section on m), and so the analysis adds ξ : m^j to CCP(x^{i-1}).

Similar to the prior write access, Raptor checks for ordering with each rd(x)^{i-1}_t by each thread t. In general, reads are not totally ordered in a –racefree execution. Thus Raptor must check for ordering between wr(x)^i and each prior read by another thread rd(x)^{i-1}_t (wr(x)^{i-1} ≺_{tr} rd(x)^{i-1}_t ≺_{tr} wr(x)^i). Lines 9–13 of Algorithm 1 add ξ to indicate PO, HB, CP, and/or CCP ordering from rd(x)^{i-1}_t by each thread t to wr(x)^i, satisfying the invariants.

Initializing sets for current access. Lines 14–15 initialize PO and HB sets for x^i. (Before this event, all sets for x^i are ∅.) In addition to adding T to PO(x^i) and HB(x^i), the analysis adds m^j to HB(x^i) for each ongoing critical section on m^j by T, satisfying the [HB-critical-section] invariant.
### 3.5.2 Handling Read Events

At a read to shared variable \( x \), i.e., \( e = \text{rd}(x)^{i}_T \), the analysis performs the actions in Algorithm 2, which is analogous to Algorithm 1. The analysis establishes CP Rule (a); checks for PO, HB, CP, and CCP order with the prior access \( \text{wr}(x)^{i} \); and initializes \( x^{i}_T \)'s sets.

#### Algorithm 2

Raptor’s analysis for \( \text{rd}(x)^{i}_T \) by \( T \)

1. for all \( m \in \text{heldBy}(T) \) do
   2. if \( \exists j \exists h \mid m^{j}_i \in \text{HB}(x^h) \land T \notin \text{PO}(x^h) \) then \( CP(m^i)^+ \leftarrow CP(m^i)^+ \cup \{ T \} \) s.t. \( j \) is max satisfying index
   3. if \( T \in \text{PO}(x^i) \) then \( PO(x^i)^+ \leftarrow PO(x^i)^+ \cup \{ \xi_T \} \)
   4. if \( T \in \text{HB}(x^i) \) then \( HB(x^i)^+ \leftarrow HB(x^i)^+ \cup \{ \xi_T \} \)
   5. if \( T \in \text{CP}(x^i) \) then \( CP(x^i)^+ \leftarrow CP(x^i)^+ \cup \{ \xi_T \} \)
   6. for all \( m \mid T : m \in \text{CCP}(x^i) \) do \( \text{CCP}(x^i)^+ \leftarrow \text{CCP}(x^i)^+ \cup \{ \xi_T : m \} \)
   7. \( PO(x^i)^+ \leftarrow \{ T \} \)
   8. \( HB(x^i)^+ \leftarrow \{ T \} \cup \{ m^j_i \mid m^j_i \in \text{heldBy}(T) \} \)
   9. \( CP(x^i)^+ \leftarrow \text{CCP}(x^i)^+ \leftarrow \emptyset \)

### Establishing Rule (a).

Lines 1–2 of Algorithm 2 add \( T \) to \( CP(m^i) \) if a prior critical section executed a conflicting write access to \( x \), establishing Rule (a), satisfying the \[[CP-rule-A]\] invariant (Figure 3.2). As we mentioned for write events, when \( T \) later releases lock \( m \), the analysis will update the \( CP(\rho) \) sets that depend on \( \text{rel}(m)^j \prec_{CP} \text{acq}(m)^k \), as Section 3.5.4 describes.

### Checking ordering with prior write access.

To represent \( \text{wr}(x)^{i} \prec_{CP} \text{rd}(x)^{i}_T \), lines 3–5 add \( \xi_T \) to each set of \( x^i \) that already contains \( T \), indicating PO, HB, CP, and/or CCP ordering from \( \text{wr}(x)^{i} \) to \( \text{rd}(x)^{i}_T \), satisfying the invariants.
Initializing sets for current read access. Lines 7–8 initialize PO and HB sets for \( x^i_T \).

In addition to adding \( T \) to \( PO(x^i_T) \) and \( HB(x^i_T) \), the analysis adds \( m^j_i \) to \( HB(x^i_T) \) for each ongoing critical section on \( m^j \) by \( T \), satisfying the \([HB\text{-critical-section}]\) invariant.

If a thread \( T \) performs multiple reads to \( x \) between \( wr(x)^i \) and \( wr(x)^{i+1} \), Raptor only needs to track sets for the latest read: if the earlier read races with \( wr(x)^{i+1} \), then so does the later read. Thus Raptor maintains \( x^i_T \)’s sets for the latest \( rd(x)^i_T \) only, which requires resetting \( x^i_T \)’s CP and CCP sets to \( \emptyset \) on each read (line 9).

3.5.3 Handling Acquire Events

At an acquire of a lock \( m \), i.e., \( e = acq(m)^i \) by thread \( T \), the analysis performs the actions in Algorithm 3. The analysis establishes HB and CP ordering from \( m \) to \( T \) for all \( \rho \); adds \( CCP(\rho) \) elements for conditionally CP (CCP) ordered critical sections; and initializes \( m^i \)’s sets.

### Algorithm 3

Raptor’s analysis for \( acq(m)^i \) by \( T \)

1: for all \( \rho \) do
   ▷ Establish order from \( m \) to \( T \)
2:   if \( m \in CP(\rho) \) then \( CP(\rho)^+ \leftarrow CP(\rho)^+ \cup \{T\} \)
3: for all \( n^k \mid m:n^k \in CCP(\rho) \) do \( CCP(\rho)^+ \leftarrow CCP(\rho)^+ \cup \{T:n^k\} \) \▷ No effect if \( \exists k' < k \mid T:n^{k'} \in CCP(\rho)^+ \)
4:   if \( \exists j \mid m^j \in HB(\rho) \) then
5:     \( HB(\rho)^+ \leftarrow HB(\rho)^+ \cup \{T\} \)
     ▷ Add CCP ordering
6:     \( CCP(\rho)^+ \leftarrow CCP(\rho)^+ \cup \{T:m^j\} \) \▷ No effect if \( \exists j' < j \mid T:m^j \in CCP(\rho)^+ \)
7:     \( PO(m^j)^+ \leftarrow \{T\} \)
8:     \( HB(m^j)^+ \leftarrow \{T\} \)

Establishing HB and CP order. Both HB and CP are closed under right-composition with HB. Thus, after the current event \( e = acq(m)^i \) by \( T \), any \( e_\rho \) that is HB or CP ordered
to m (i.e., \( e_\rho \prec_{\text{HB}} \text{rel}(m)^j \) or \( e_\rho \prec_{\text{CP}} \text{rel}(m)^j \) for some \( j < i \)) is now also HB or CP ordered, respectively, to T. Specifically, if \( m \in CP(\rho) \), then \( e_\rho \prec_{\text{CP}} \text{acq}(m)^i \) and the analysis adds T to \( CP(\rho) \) (line 2), satisfying the [CP] invariant. Similarly, lines 4–5 establishes HB order from m to T, satisfying the [HB] invariant. For the condition at line 4, recall that \( m^i \in HB(\rho) \) if \( m^i \in HB(\rho) \).

**Establishing CCP order.** The analysis establishes CCP order at line 3. For any critical section on lock \( n^k \), if \( m : n^k \in CCP(\rho) \), then \( e_\rho \prec_{\text{CP}} \text{rel}(m)^{i-1} \) if \( \text{acq}(n)^k \prec_{\text{CP}} \text{rel}(n)^j \) where \( n^j \) is an ongoing critical section. After the current event \( \text{acq}(m)^i \), since HB right-composes with CP, \( e_\rho \prec_{\text{CP}} \text{acq}(m)^i \) if \( \text{acq}(n)^k \prec_{\text{CP}} \text{rel}(n)^j \). Thus, the analysis adds T to \( CCP(\rho) \).

Additionally, if \( e_\rho \prec_{\text{HB}} \text{rel}(m)^j \) for some \( j < i \), then by Rule (b), \( e_\rho \prec_{\text{CP}} \text{acq}(m)^i \) if \( \text{acq}(m)^j \prec_{\text{CP}} \text{rel}(m)^j \). Line 6 handles this case by adding \( T : m^i \) to \( CCP(\rho) \) when \( e_\rho \prec_{\text{HB}} \text{rel}(m)^j \), satisfying the [CCP-constraint] invariant.

**Initializing sets for current lock.** Lines 7–8 initialize \( m^i \)’s sets. The analysis adds T to \( PO(m^i) \) and \( HB(m^i) \), since \( m^i \) will be PO and HB ordered to any event by T after \( \text{acq}(m)^i \), satisfying the [PO] and [HB] invariants. (The analysis never needs or uses \( PO(m^i) \). Raptor adds T to \( PO(m^i) \) only to satisfy Figure 3.2’s [PO] invariant.)

### 3.5.4 Handling Release Events

At a lock release, i.e., \( e = \text{rel}(m)^i \) by thread T, the analysis performs the actions in Algorithm 4, called the “pre-release” algorithm, followed by the actions in Algorithm 5, called the “release” algorithm. We divide Raptor’s analysis into two algorithms to separate the changes to \( CCP(\rho) \) elements: the pre-release algorithm adds elements to \( CP(\rho) \) and \( CCP(\rho) \), and the release algorithm uses the updated sets.
3.5.4.1 Pre-release Algorithm (Algorithm 4)

Due to Rule (b), whether \( e_{\rho} \) is CP ordered to some lock or thread can depend on whether \( \text{acq}(m)^j \prec_{\text{CP}} \text{rel}(m)^i \) (for some \( j \)). The pre-release algorithm establishes Rule (b) by updating CP and CCP sets that depend on whether \( \text{acq}(m)^j \prec_{\text{CP}} \text{rel}(m)^i \). Lines 2–3 establish Rule (b) directly by detecting that \( \text{acq}(m)^j \prec_{\text{CP}} \text{rel}(m)^i \), then updating \( CP(\rho) \) sets for every dependent element \( \sigma \), satisfying the \([\text{CP}]\) invariant.

<table>
<thead>
<tr>
<th>Algorithm 4</th>
<th>Raptor’s analysis for pre-release of ( \text{rel}(m)^i ) by ( T )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: for all ( \rho ) do</td>
<td></td>
</tr>
<tr>
<td>\quad \triangleright \text{Handle CCP ordering according to Rule (b)}</td>
<td></td>
</tr>
<tr>
<td>2: for all ( \sigma, j \mid \sigma : m^j \in CP(\rho) ) do</td>
<td></td>
</tr>
<tr>
<td>\quad \text{if } \exists l \geq j \mid T \in CP(m^l) \text{ then } CP(\rho)^+ \leftarrow CP(\rho)^+ \cup {\sigma}</td>
<td></td>
</tr>
<tr>
<td>\quad \triangleright \text{Transfer CCP to depend on other lock(s)}</td>
<td></td>
</tr>
<tr>
<td>3: for all ( n^k \mid (\exists l \geq j \mid T : n^k \in CP(m^l)) ) do</td>
<td></td>
</tr>
<tr>
<td>\quad CCP(\rho)^+ \leftarrow CCP(\rho)^+ \cup {\sigma : n^k}</td>
<td></td>
</tr>
<tr>
<td>\quad \triangleright \text{No effect if } \exists k' &lt; k \mid \sigma : n^{k'} \in CP(\rho)^+</td>
<td></td>
</tr>
</tbody>
</table>

Even if \( \text{acq}(m)^j \prec_{\text{CP}} \text{rel}(m)^i \), it may not be knowable at \( \text{rel}(m)^i \) because it may be dependent on if \( \text{acq}(n)^k \prec_{\text{CP}} \text{rel}(n)^h \) (where \( n^h \) is the ongoing critical section on \( n \)). Lines 4–5 detect such cases and “transfer” the CCP dependence from \( m \) to \( n \). This transfer is necessary because \( m \)’s critical section is ending, and the release algorithm will remove all \( \sigma : m^j \) elements. After the pre-release algorithm, Figure 3.2’s invariants still hold—for the point in time prior to \( e \).

\(^7\)More precisely, the analysis checks for any \( l \geq j \) \( \text{acq}(m)^l \prec_{\text{CP}} \text{rel}(m)^i \), since HB left-composes with CP.
3.5.4.2 Release Algorithm (Algorithm 5)

The release algorithm operates on the analysis state modified by the pre-release algorithm. The analysis establishes CP and HB order from T to m; and removes all CCP elements that are dependent on m.

Algorithm 5

Raptor’s analysis for \( rel(m)^i \) by T

1: for all \( \rho \) do
   ▷ Establish order from T to m
2:   if \( T \in CP(\rho) \) then \( CP(\rho)^+ \leftarrow CP(\rho)^+ \cup \{m\} \)
3:   for all \( n^k \mid n \neq m \land T:n^k \in CCP(\rho) \) do \( CCP(\rho)^+ \leftarrow CCP(\rho)^+ \cup \{m:n^k\} \)
4:   if \( T \in HB(\rho) \) then \( HB(\rho)^+ \leftarrow HB(\rho)^+ \cup \{m^i\} \)
   ▷ No effect if \( \exists i' < i \mid m'^i \in HB(\rho)^+ \)
5:   \( CCP(\rho)^+ \leftarrow CCP(\rho)^+ \setminus \{\sigma : m^j \in CCP(\rho)\} \)

Since HB and CP are closed under right-composition with HB, if \( e_\rho \) is HB or CP ordered to T before \( rel(m)^i \), then \( e_\rho \) is HB or CP ordered to m after \( rel(m)^i \). The analysis establishes order from T to m (lines 2–4), satisfying the [HB] and [CP] invariants. This is analogous to lines 2–6 of Algorithm 3’s establishing order from m to T at \( acq(m)^i \). Line 3 establishes CCP order from T to m for any lock instance \( n^k \ (n \neq m) \) such that \( T:n^k \in CCP(\rho) \) similar to Algorithm 3’s line 3.

Line 5 removes all CCP elements dependent on m, i.e., all \( \sigma : m^j \) elements from \( CCP(\rho) \), satisfying the [CCP-constraint] invariant. Removal is necessary: it would be incorrect for the analysis to retain these elements, e.g., \( acq(m)^j \prec_{CP} rel(m)^{i+1} \) does not imply that \( e_\rho \) is CP ordered to \( \sigma \). It is safe to remove all \( \sigma : m^j \) elements, even if \( acq(m)^j \prec_{CP} rel(m)^i \) is not knowable at \( e = rel(m)^i \), because the pre-release algorithm has already handled transferring all such \( \sigma \) whose CCP order depends on a lock other than m.

After the release algorithm, Figure 3.2’s invariants hold—for the point in time after e.
3.5.5 Removing Obsolete Sets and Detecting CP-Races

Raptor maintains sets for every variable access and lock acquire. Without removing sets, the analysis state’s size would be proportional to trace length, which—since the analysis iterates over all non-empty set owners at acquire and release events—would be unscalable in terms of both space and time. Fortunately, for real (non-adversarial) program executions, most set owners become obsolete—meaning that they will not be needed again—relatively quickly. Raptor detects obsolete set owners and removes each owner’s sets, saving both space and time.

Removing obsolete variable access sets and detecting CP-races. A variable access set owner \( x^i \) or \( x^i_T \) becomes obsolete once the analysis determines whether or not the corresponding access (\( wr(x)^i \) or \( rd(x)^i_T \), respectively) is involved in a CP-race with the next access. Detecting CP-races is thus naturally part of checking for obsolete set owners.

Algorithm 6 shows the conditions for determining whether \( x^i \) or \( x^i_T \) is obsolete or has a CP-race with another access to variable \( x \). For write owner \( x^i \), if Raptor has determined that \( wr(x)^i \prec_{CP} wr(x)^{i+1} \) or \( wr(x)^i \prec_{PO} wr(x)^{i+1} \), then according to the [PO] and [CP] invariants (Figure 3.2), \( \xi \in CP(x^i) \cup PO(x^i) \). For Raptor to later determine that \( wr(x)^i \prec_{CP} wr(x)^{i+1} \), according to the [CP] and [CCP-constraint] invariants there must be some \( m^j \) such that \( \xi : m^j \in CCP(x^i) \). Line 2 checks these conditions and reports a race if Raptor has determined PO and CP order between the conflicting write accesses has not and will not be established. Similarly, lines 4 and 5 check the conditions for reporting write–read and read–write CP-races, respectively.

After \( wr(x)^{i+1} \) has executed, \( x^i \) or \( x^i_T \) is obsolete if Raptor definitely will not in the future determine that \( wr(x)^i \) or \( rd(x)^i_T \), respectively, is CP ordered to a following conflicting
access. To determine whether future CP ordering is possible, according to the [CP] and [CCP-constraint] invariants there must exist a $\xi : m^i$ or $\xi_T : m^i$ in $CCP(x^i)$ or $CCP(x^i_T)$.

Lines 6–8 check these conditions, and remove obsolete sets for $x^i$ and $x^i_T$.

### Algorithm 6  Detect obsolete owners and remove sets and report CP-races for $x^i$ and $x^i_T$ (for all threads $t$)

1: if $wr(x)^{i+1}$ has executed then
2:   if $\xi \notin CP(x^i) \cup PO(x^i) \land \nexists m^i \mid \xi : m^i \in CCP(x^i)$ then Report CP-race on $wr(x)^i$ and $wr(x)^{i+1}$
3:   for all threads $t$ such that $rd(x)^i_t$ executed do
4:     if $\xi_t \notin CP(x^i_t) \cup PO(x^i_t) \land \exists m^i \mid \xi : m^i \in CCP(x^i_t)$ then Report CP-race on $rd(x)^i_t$ and $wr(x)^{i+1}$
5:     if $\exists m^i \mid \xi : m^i \in CCP(x^i_t)$ then Remove $PO(x^i_t)^+, HB(x^i_t)^+, CP(x^i_t)^+, CCP(x^i_t)^+$
6:   if $\exists m^i \mid \xi : m^i \in CCP(x^i) \land \forall$ threads $t$ $\exists m^i \mid \xi : m^i \in CCP(x^i)$ then
7:     Remove $PO(x^i)^+, HB(x^i)^+, CP(x^i)^+, CCP(x^i)^+$  $\triangleright$ No CP-race (unless reported above)

When the execution terminates (i.e., after the last event in the observed total order), we assume that no thread holds any lock,\(^8\) so the CCP sets for all owners are empty by the [CCP-constraint] invariant (Figure 3.2). Thus for every $wr(x)^i \succeq wr(x)^{i+1}$, $wr(x)^i \succeq rd(x)^i_t$, and $rd(x)^i_t \succeq wr(x)^{i+1}$ pair for which Raptor has not already ruled out a (-racei.e., $\xi \in CP(x^i) \cup PO(x^i)$, $\xi_t \in CP(x^i_t) \cup PO(x^i_t)$, and $\xi \in CP(x^i_t) \cup PO(x^i_t)$, respectively), Raptor eventually reports a CP-race.

### 3.5.6 Correctness of detecting CP-races

Now that we know how Raptor detects CP-races, we can prove that Raptor is a sound and complete CP-race detector.

**Theorem 2.** An execution has a CP-race if and only if Raptor reports a CP-race for the execution.

**Proof.** We prove the forward direction ($\Rightarrow$) and backward direction ($\Leftarrow$) in turn:

\(^8\)If an execution does not satisfy this condition, Raptor can simulate the release of all held locks, by performing the pre-release and release algorithms (Algorithms 4 and 5) for each held lock.

48
**Forward direction (completeness).** Suppose a trace $tr$ has a CP-race, but Raptor does not report a CP-race. Without loss of generality, let $e$ and $e'$ be access (read/write) events such that $e <_{tr} e'$ and $e \simeq e'$. Let $\rho$ be the set owner for event $e$ (e.g., $x^i$ for $\text{wr}(x)^i$), and $\xi_*$ be $\xi$ or $\xi_T$ depending on whether $e'$ is a write or a read by $T$, respectively. Then $e \not<_{\text{cp}} e' \land e \not<_{\text{po}} e'$, but $\xi_* \in CP(\rho) \cup PO(\rho)$ at program termination. According to Theorem 1, by the [CP] and [PO] invariants, $\xi_* \notin CP(\rho) \land \xi_* \notin PO(\rho)$ at program termination, which is a contradiction.

**Backward direction (soundness).** Suppose Raptor reports a CP-race between access events $e$ and $e'$ such that $e <_{tr} e'$, but no CP-race exists. Let $\rho$ be the set owner for event $e$ (e.g., $x^i$ for $\text{wr}(x)^i$), and $\xi_*$ be $\xi$ or $\xi_T$ depending on whether $e'$ is a write or a read by $T$, respectively. Then $\xi_* \notin CP(\rho) \cup PO(\rho)$ at program termination, but $e <_{\text{cp}} e' \lor e <_{\text{po}} e'$.

Since $\xi_* \notin PO(\rho)$ at program termination, according to Theorem 1, by the [PO] invariant, $e \not<_{\text{cp}} e'$. Thus $e <_{\text{cp}} e'$. By the [CP] invariant, at program termination,

$$\xi_* \in CP(\rho) \lor (\exists n^k | \xi_* : n^k \in CCP(\rho) \land \exists j | \text{rel}(n)^j <_{\text{cp}} \text{acq}(n)^j <_{tr} e^\Omega)$$

where $e^\Omega$ represents a final “termination” event. Since $\xi_* \notin CP(\rho)$, $\exists n^k | \xi_* : n^k \in CCP(\rho)$.

By the [CCP-constraint] invariant, at $e^\Omega$, $\exists l | \text{acq}(n)^l <_{tr} e^\Omega \land \text{rel}(n)^l \not<_{tr} e^\Omega$. However, an execution releases all held locks before terminating, so $\text{rel}(n)^l <_{tr} e^\Omega$, which is a contradiction.

**Removing obsolete lock acquire sets.** Raptor uses sets for lock instances, such as $CP(m^j)$, for detecting CP-ordered critical sections and tracking CP ordering from $\text{acq}(m)^j$ to establish Rule (b). Once $m^j$’s sets’ elements can no longer trigger Rule (b), $m^j$ is obsolete.

Algorithm 7 shows the condition for whether $m^j$ is obsolete. Lock owner $m^j$ is not obsolete if any set owner’s $CCP(\rho)$ set contains $\sigma : m^i (i \leq j)$ or might contain it at some later
event (indicated by \( m^j \) being in \( HB(\rho) \))—unless \( m \in CP(\rho) \), in which case \( \sigma : m^j \in CCP(\rho) \) would be superfluous. Line 2 shows the exact condition; if it evaluates to true, then the pre-release algorithm (Algorithm 4) definitely will not use \( m^j \)'s sets anymore, and so the removal algorithm removes each set.

### Algorithm 7

Detect obsolete owner and remove sets for \( m^j \)

1: if rel(m)^j has executed then
2:   if \( \exists \rho \mid \rho \neq m^j \land (m^j \in HB(\rho) \lor m^j \in HB(\rho) \lor (\exists i \leq j \mid \sigma : m^i \in CCP(\rho))) \land m \notin CP(\rho) \) then
3:     Remove PO(m^j)^+, HB(m^j)^+, CP(m^j)^+, CCP(m^j)^+

### 3.6 Examples

Figure 3.4 extends the example from Figure 3.1 with Raptor’s analysis state after each event. At \( acq(m)^2 \), Raptor adds \( T2 : m^1 \) to all \( CCP(\rho) \) (line 6 in Algorithm 3) such that \( e_\rho \prec_{hb} rel(m)^1 \) (line 4 in Algorithm 3), since \( e_\rho \prec_{cp} acq(m)^2 \) if \( acq(m)^1 \prec_{cp} rel(m)^2 \).

At \( rd(x)^1_{T3} \), Raptor adds \( \xi_{T3} : m^1 \) to \( CCP(x^1) \) (line 6 in Algorithm 2), capturing that \( wr(x)^1 \prec_{cp} rd(x)^1_{T3} \) if \( rel(m)^1 \prec_{cp} acq(m)^2 \). However, it is not knowable at this point that \( wr(x)^1 \prec_{cp} rd(x)^1_{T3} \). At \( rd(y)^1_{T2} \), Raptor establishes Rule (a) by adding \( T2 \) to \( CP(u^1) \) (line 2 in Algorithm 2). Although it is possible to infer \( wr(x)^1 \prec_{cp} rd(x)^1_{T3} \) at this point, Raptor defers this logic until \( rel(m)^2 \), when the pre-release algorithm (Algorithm 4) detects \( acq(m)^1 \prec_{cp} rel(m)^2 \) and handles CCP elements of the form \( \sigma : m^1 \). At \( rel(m)^2 \), since \( T2 \in CP(m^1) \) (line 3 in Algorithm 4) and \( \xi_{T3} : m^1 \in CCP(x^1) \) (line 2 in Algorithm 4), Raptor adds \( \xi_{T3} \) to \( CP(x^1) \) (line 3 in Algorithm 4), indicating \( wr(x)^1 \prec_{cp} rd(x)^1_{T3} \).

Figure 3.5 shows a more complex execution requiring “transfer” of CCP ordering, in which \( wr(x)^1 \prec_{cp} wr(x)^2 \) because \( acq(m)^1 \prec_{cp} rel(m)^2 \), which in turn depends on \( acq(o)^1 \prec_{cp} rel(o)^2 \). Even when \( m^2 \)'s critical section ends at \( rel(m)^2 \), it is not knowable that \( acq(m)^1 \prec_{cp}
rel(m)^2. At \( \text{rel}(m)^2 \), the pre-release algorithm “transfers” CCP ordering from \( m \) to \( o \) it adds \( p:o^1 \) to \( CCP(x^1) \) (line 5 in Algorithm 4) because \( T3:o^1 \in CCP(m^1) \) (line 4 in Algorithm 4) and \( p:m^1 \in CCP(x^1) \) (line 2 in Algorithm 4). As a result, at \( \text{wr}(x)^2 \), Raptor adds \( \xi:o^1 \) to \( CCP(x^1) \) (line 8 in Algorithm 1). Finally, at \( \text{rel}(o)^2 \), the analysis adds \( \xi \) to \( CP(x^1) \) (line 3 in Algorithm 4) because \( T2 \in CP(o^1) \) (line 3 in Algorithm 4) and \( \xi:o^1 \in CCP(x^1) \) (line 2 in Algorithm 4).

Figure 3.6 presents an even more complex execution involving “transfer” of CCP ordering, in which \( \text{wr}(x)^1 \prec_{cp} \text{wr}(x)^2 \) because \( \text{acq}(m)^1 \prec_{cp} \text{rel}(m)^2 \), which in turn depends on \( \text{rel}(q)^1 \prec_{cp} \text{acq}(q)^2 \).

At event \( \text{rel}(m)^2 \), an online analysis cannot determine that \( \text{acq}(m)^1 \prec_{cp} \text{rel}(m)^2 \) because it is not knowable that \( \text{rel}(q)^1 \prec_{cp} \text{acq}(q)^2 \). At \( \text{rel}(m)^2 \), Raptor’s pre-release algorithm “transfers” CCP ordering from \( m \) to \( q \) by adding \( T5:q^1 \) to \( CCP(x^1) \) (line 5 in Algorithm 4) because \( T4:q^1 \in CCP(m^1) \) (line 4 in Algorithm 4) and \( T5:m^1 \in CCP(x^1) \) (line 2 in Algorithm 4). As a result, at \( \text{wr}(x)^2 \), the analysis thus adds \( \xi:q^1 \) to \( CCP(x^1) \) (line 8 in Algorithm 1). Finally, at \( \text{rel}(q)^2 \), Raptor adds \( \xi \) to \( CP(x^1) \) (line 3 in Algorithm 4) because \( T3 \in CP(q^1) \) (line 3 in Algorithm 4) and \( T5:q^1 \in CCP(x^1) \) (line 2 in Algorithm 4).

Alternatively, suppose that thread \( T1 \) executed its critical section on \( o \) before its critical section on \( m \). In that subtly different execution, \( \text{wr}(x)^1 \not\prec_{cp} \text{wr}(x)^2 \). Raptor tracks analysis state that achieves capturing the difference between these two execution variants.
Figure 3.4: The execution from Figure 3.1, in which \( wr(x)^1 \prec_{CP} rd(x)^1 \), with Raptor’s analysis state updates shown in the same format as Figure 3.3. For brevity, this figure and the article’s remaining figures omit showing updates to \( PO(\rho) \).
Figure 3.5: An execution illustrating CCP transfer in which $wr(x)^1 \prec_{cp} wr(x)^2$, in the same format as Figure 3.3. For space, the figure omits set updates for $x^2$ and $y^2$. 
<table>
<thead>
<tr>
<th>Execution</th>
<th>Analysis state changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1 T2 T3 T4 T5</td>
<td>s¹ y¹ m¹ q¹ q² p¹ r¹ m² o² q³ p² r²</td>
</tr>
<tr>
<td>acq(m)¹</td>
<td></td>
</tr>
<tr>
<td>wr(x)¹</td>
<td></td>
</tr>
<tr>
<td>acq(m)²</td>
<td></td>
</tr>
<tr>
<td>rel(r)²</td>
<td></td>
</tr>
<tr>
<td>acq(q)²</td>
<td></td>
</tr>
<tr>
<td>rel(q)²</td>
<td></td>
</tr>
<tr>
<td>acq(p)²</td>
<td></td>
</tr>
<tr>
<td>rel(p)²</td>
<td></td>
</tr>
<tr>
<td>acq(r)²</td>
<td></td>
</tr>
<tr>
<td>rel(r)²</td>
<td></td>
</tr>
<tr>
<td>acq(p)²</td>
<td></td>
</tr>
<tr>
<td>rel(p)²</td>
<td></td>
</tr>
<tr>
<td>acq(r)²</td>
<td></td>
</tr>
<tr>
<td>rel(r)²</td>
<td></td>
</tr>
<tr>
<td>acq(p)²</td>
<td></td>
</tr>
<tr>
<td>rel(p)²</td>
<td></td>
</tr>
<tr>
<td>acq(r)²</td>
<td></td>
</tr>
<tr>
<td>rel(r)²</td>
<td></td>
</tr>
<tr>
<td>acq(p)²</td>
<td></td>
</tr>
<tr>
<td>rel(p)²</td>
<td></td>
</tr>
<tr>
<td>acq(r)²</td>
<td></td>
</tr>
<tr>
<td>rel(r)²</td>
<td></td>
</tr>
<tr>
<td>acq(p)²</td>
<td></td>
</tr>
<tr>
<td>rel(p)²</td>
<td></td>
</tr>
<tr>
<td>acq(r)²</td>
<td></td>
</tr>
<tr>
<td>rel(r)²</td>
<td></td>
</tr>
<tr>
<td>acq(p)²</td>
<td></td>
</tr>
<tr>
<td>rel(p)²</td>
<td></td>
</tr>
<tr>
<td>acq(r)²</td>
<td></td>
</tr>
<tr>
<td>rel(r)²</td>
<td></td>
</tr>
<tr>
<td>acq(p)²</td>
<td></td>
</tr>
<tr>
<td>rel(p)²</td>
<td></td>
</tr>
<tr>
<td>acq(r)²</td>
<td></td>
</tr>
<tr>
<td>rel(r)²</td>
<td></td>
</tr>
<tr>
<td>acq(p)²</td>
<td></td>
</tr>
<tr>
<td>rel(p)²</td>
<td></td>
</tr>
<tr>
<td>acq(r)²</td>
<td></td>
</tr>
<tr>
<td>rel(r)²</td>
<td></td>
</tr>
<tr>
<td>acq(p)²</td>
<td></td>
</tr>
<tr>
<td>rel(p)²</td>
<td></td>
</tr>
<tr>
<td>acq(r)²</td>
<td></td>
</tr>
<tr>
<td>rel(r)²</td>
<td></td>
</tr>
<tr>
<td>acq(p)²</td>
<td></td>
</tr>
<tr>
<td>rel(p)²</td>
<td></td>
</tr>
<tr>
<td>acq(r)²</td>
<td></td>
</tr>
<tr>
<td>rel(r)²</td>
<td></td>
</tr>
<tr>
<td>acq(p)²</td>
<td></td>
</tr>
<tr>
<td>rel(p)²</td>
<td></td>
</tr>
<tr>
<td>acq(r)²</td>
<td></td>
</tr>
<tr>
<td>rel(r)²</td>
<td></td>
</tr>
<tr>
<td>acq(p)²</td>
<td></td>
</tr>
<tr>
<td>rel(p)²</td>
<td></td>
</tr>
<tr>
<td>acq(r)²</td>
<td></td>
</tr>
<tr>
<td>rel(r)²</td>
<td></td>
</tr>
<tr>
<td>acq(p)²</td>
<td></td>
</tr>
<tr>
<td>rel(p)²</td>
<td></td>
</tr>
<tr>
<td>acq(r)²</td>
<td></td>
</tr>
<tr>
<td>rel(r)²</td>
<td></td>
</tr>
<tr>
<td>acq(p)²</td>
<td></td>
</tr>
<tr>
<td>rel(p)²</td>
<td></td>
</tr>
<tr>
<td>acq(r)²</td>
<td></td>
</tr>
<tr>
<td>rel(r)²</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.6: A more complex execution than Figure 3.5 showing CCP transfer, where wr(x)¹ $\prec_{cp}$ wr(x)². For space, the figure omits set updates for x² and y².
3.7 Correctness: Raptor Tracks CP Soundly and Completely

This section proves that Raptor soundly and completely tracks CP, by proving Theorem 1: that Raptor maintains Figure 3.2’s invariants after every event. For brevity, our proof only shows that Raptor maintains the [CP] invariant. For the other invariants, it is comparatively straightforward to see that they hold:

- Raptor maintains HB sets in much the same way as Goldilocks [26], which provably computes HB soundly and completely [25]. Thus it is fairly straightforward to see that Raptor maintains the [PO], [HB], [HB-index], and [HB-critical-section] invariants.

- The analysis maintains the [CP-rule-A] invariant by updating CP sets as soon as a conflicting write or read executes (lines 1–4 of Algorithm 1 and lines 1–2 of Algorithm 2).

- The [CCP-constraint] invariant holds because CCP elements of the form \( \sigma : m^i \) only exist during critical sections on \( m \), by the following argument. At \( \text{rel}(m)^j \), Algorithm 5 removes all elements \( \sigma : m^i \) from every \( \text{CCP}(\rho)^+ \). When \( \not\exists \sigma' \mid \sigma' : m^i \in \text{CCP}(\rho) \), only Algorithm 3 (for \( \text{acq}(m)^j \)) adds elements of the form \( \sigma : m^i \) to \( \text{CCP}(\rho)^+ \).

We first prove two lemmas that we will use to prove Theorem 1:

**Lemma 1.** Let \( e \) be any release event, i.e., \( e = \text{rel}(m)^j \) by thread \( T \). If Figure 3.2’s invariants hold before Raptor’s pre-release algorithm (Algorithm 4) executes, then after the pre-release algorithm executes, the invariants still hold, for the moment in time before \( e \) executes.

**Proof.** Let \( e = \text{rel}(m)^j \) by thread \( T \). Let \( \rho \) be any set owner. Let \( e_\rho \) be the event corresponding to \( \rho \), i.e., \( e_\rho = \text{wr}(x)^h \) if \( \rho = x^h \), or \( e_\rho = \text{acq}(m)^h \) if \( \rho = m^h \).

55
We define the following abbreviations for the left- and right-hand sides of the \([\text{CP}]\) invariant:

Let \(LHS = CP(\rho) \cup \{ \sigma \mid (\exists n^k \mid \sigma : n^k \in CCP(\rho) \land \exists l | \text{rel}(n)^k \prec_{\text{cp}} \text{acq}(n)^l <_{tr} e) \} \).

Let \(LHS^+ = CP(\rho)^+ \cup \{ \sigma \mid (\exists n^k \mid \sigma : n^k \in CCP(\rho)^+ \land \exists l | \text{rel}(n)^k \prec_{\text{cp}} \text{acq}(n)^l <_{tr} e) \} \).

Let \(RHS = \{ \sigma \mid (\exists e' | \text{appl}(\sigma, e') \land e_p \prec_{\text{cp}} e' <_{tr} e) \} \).

Suppose Figure 3.2’s invariants hold (i.e., \(LHS = RHS\)) before the pre-release algorithm executes.

To show \(LHS^+ = RHS\), we show \(LHS^+ \subseteq RHS\) (subset) and \(LHS^+ \supseteq RHS\) (superset) in turn.

**Subset direction:** Let \(\sigma \in LHS^+\).

Either \(\sigma \in CP(\rho)^+\) or \(\exists n^k \mid \sigma : n^k \in CCP(\rho)^+ \land \exists l | \text{rel}(n)^k \prec_{\text{cp}} \text{acq}(n)^l <_{tr} e\) (or both):

**Case 1:** \(\sigma \in CP(\rho)^+\)

If \(\sigma \in CP(\rho)\), then \(\sigma \in LHS\), so \(\sigma \in RHS\) because \(LHS = RHS\).

Otherwise (\(\sigma \notin CP(\rho)\)), Algorithm 4 adds \(\sigma\) to \(CP(\rho)^+\), which can happen only at line 3. Let \(j\) be such that \(\sigma : m^j \in CCP(\rho)\) (line 2) and \(T \in CP(m^q) \mid q \geq j\) (line 3). Since \(T \in CP(m^q)\) and \(LHS = RHS\), therefore \(T \in RHS\). Thus \(\exists e' | thr(e') = T \land \text{acq}(m)^q \prec_{\text{cp}} e' <_{tr} e\). Since \(e' <_{tr} e\) and \(thr(e') = thr(e)\), \(e' \prec_{HB} e\) by the definition of HB. Thus \(\text{acq}(m)^q \prec_{cp} e\) by CP Rule (c) and therefore \(\text{rel}(m)^q \prec_{cp} \text{acq}(m)^i\) by CP Rule (b) (recall that \(e = \text{rel}(m)^i\)). If \(q = j\), then \(\text{rel}(m)^j \prec_{cp} \text{acq}(m)^i\). Otherwise, \(q > j\) so \(\text{acq}(m)^j \prec_{HB} \text{acq}(m)^q\) by the definition of HB. Thus \(\text{acq}(m)^j \prec_{cp} e\) and therefore \(\text{rel}(m)^j \prec_{cp} \text{acq}(m)^i\). Since \(\sigma : m^j \in CCP(\rho)\) and \(\text{rel}(m)^j \prec_{cp} \text{acq}(m)^i\), \(\sigma \in LHS\) and thus \(\sigma \in RHS\).
Lemma 2. Let \( e \) be any release event, i.e., \( e \overset{\text{rel}}{\longrightarrow} \text{acq}(m) \) such that

\[
\text{Definition.} \quad \text{The CP-distance measures the number of times that Rule (b) and Rule (c) must be applied for a CP ordering despite removing CCP set elements. To do so, we define a concept called CP-distance that measures the number of times that Rule (b) and Rule (c) must be applied for a CP ordering between two critical sections on the same lock ordered by CP:}
\]

Superset direction: Let \( \sigma : m^i \in CCP(\rho) \land rel(n)^j \overset{\text{cp}}{\prec} \text{acq}(n)^j \overset{\text{tr}}{\prec} e \).

If \( \sigma : n^k \in CCP(\rho) \), then \( \sigma \in LHS \), so \( \sigma \in RHS \) because \( LHS = RHS \).

Otherwise, \( \neg (\sigma : n^k \in CCP(\rho) \land rel(n)^k \overset{\text{cp}}{\prec} \text{acq}(n)^j \overset{\text{tr}}{\prec} e) \). We know \( \text{rel}(n)^k \overset{\text{cp}}{\prec} \text{acq}(n)^j \overset{\text{tr}}{\prec} e \), so therefore \( \sigma : n^k \notin CCP(\rho) \). Thus Algorithm 4 adds \( \sigma : n^k \) to \( CCP(\rho) \) only at line 5. Let \( j \) be such that \( \sigma : m^j \in CCP(\rho) \) (line 2) and \( T : n^k \in CCP(m^q) \mid q \geq j \) (line 4). Since \( T : n^k \in CCP(m^q) \), \( \text{rel}(n)^k \overset{\text{cp}}{\prec} \text{acq}(n)^j \overset{\text{tr}}{\prec} e \), and since \( LHS = RHS \), therefore \( T \in RHS \). Thus \( \exists e' \mid \text{thr}(e') = T \land \text{acq}(m)^j \overset{\text{cp}}{\prec} e' \overset{\text{tr}}{\prec} e \). Since \( e' \overset{\text{tr}}{\prec} e \) and \( \text{thr}(e') = \text{thr}(e) \), \( e' \overset{\text{hb}}{\prec} e \) by the definition of HB. Thus \( \text{acq}(m)^q \overset{\text{cp}}{\prec} e \) by CP Rule (c) and therefore \( \text{rel}(m)^j \overset{\text{cp}}{\prec} \text{acq}(m)^j \) by CP Rule (b) (recall \( e = \text{rel}(m)^j \)). If \( q = j \), then \( \text{rel}(m)^j \overset{\text{cp}}{\prec} \text{acq}(m)^j \). Otherwise, \( q > j \) so \( \text{acq}(m)^j \overset{\text{hb}}{\prec} \text{acq}(m)^q \) by the definition of HB. Thus \( \text{acq}(m)^j \overset{\text{cp}}{\prec} e \) and therefore \( \text{rel}(m)^j \overset{\text{cp}}{\prec} \text{acq}(m)^j \). We have thus determined that \( \sigma : m^i \in CCP(\rho) \land \text{rel}(m)^j \overset{\text{cp}}{\prec} \text{acq}(m)^j \overset{\text{tr}}{\prec} e \), i.e., \( \sigma \in LHS \) and thus \( \sigma \in RHS \).

**Superset direction:** Let \( \sigma \in RHS \). Since \( LHS = RHS \), \( \sigma \in LHS \). Since Algorithm 4 maintains \( CP(\rho)^+ \supseteq CP(\rho) \) and \( CCP(\rho)^+ \supseteq CCP(\rho) \), therefore \( \sigma \in LHS^+ \).

Next we introduce and prove a lemma that helps to show that Raptor maintains invariants despite removing CCP set elements. To do so, we define a concept called \( CP-distance \) that measures the number of times that Rule (b) and Rule (c) must be applied for a CP ordering between two critical sections on the same lock ordered by CP:

**Definition.** The CP-distance of \( m^j \) and \( m^i \), \( d(m^i \rightsquigarrow m^j) \), is defined as follows:

\[
d(m^i \rightsquigarrow m^j) = \begin{cases} 
0 & \text{if } \exists e', i', j' : j \leq j' < i' \land e \not\equiv e' \land \text{acq}(m)^{i'} \overset{\text{po}}{\prec} e \overset{\text{po}}{\prec} \text{rel}(m)^{j'} \land \text{acq}(m)^{j'} \overset{\text{po}}{\prec} e' \overset{\text{po}}{\prec} \text{rel}(m)^{j'} \\
1 + \min d(n^k \rightsquigarrow n') \mid \text{acq}(m)^j \overset{\text{hb}}{\prec} \text{rel}(n)^k \overset{\text{cp}}{\prec} \text{acq}(n)^j \overset{\text{hb}}{\prec} \text{rel}(m)^j \text{ otherwise}
\end{cases}
\]

**Lemma 2.** Let \( e \) be any release event, i.e., \( e = \text{rel}(m)^j \) executed by thread \( T \). If there exists \( j \) such that \( \text{acq}(m)^j \overset{\text{cp}}{\prec} e \), and Figure 3.2’s invariants hold for all execution points up to and
including the point immediately before e, then after the pre-release algorithm (Algorithm 4) executes but before the release algorithm (Algorithm 5) executes, one or both of the following hold:

- ∃ j' ≥ j | T ∈ CP(m')
- ∃ j' ≥ j | ∃ n^k | n ≠ m ∧ ∃ l | rel(n^k) ≺_{CP} acq(n)^j ≺_{tr} e ∧ d(n^k ⇝ n^l) < d(m^j ⇝ m')

The intuition here is that if acq(m)^j ≺_{CP} e but T is not in some CP(m') (j' ≥ j), then there exists some n^k which has lesser CP-distance than m^j such that acq(m)^j ≺_{CP} e is CCP dependent on acq(n)^k ≺_{CP} rel(n)^l (where n^l is an ongoing critical section). As a result, the pre-release algorithm can safely transfer any CCP ordering dependent on m^j to be dependent on n^k, preserving invariants.

Proof. Let e = rel(m)^i executed by thread T. Let j be such that acq(m)^j ≺_{CP} e. Suppose Figure 3.2’s invariants hold for all execution points up to and including the point immediately before e.

We prove the lemma by induction on the CP-distance d(m^j ⇝ m^i).

Base case: If d(m^j ⇝ m^i) = 0, then by the definition of CP-distance, ∃ x, g, h, i', j' | j ≤ j' < i' ≤ i ∧ acq(m)^j' ≺_{po} wr(x)^g ≺_{po} rel(m)^j' ∧ acq(m)^i' ≺_{po} wr(x)^h ≺_{po} rel(m)^i'.

By the [CP-rule-A] invariant, T' ∈ CP(m') at rel(m)^i', where T' = thr(rel(m)^i'). Because rel(m)^j' ≺_{hb} rel(m)^i and the algorithms propagater HB-ordered events through CP locksets, T ∈ CP(m') at e.

Inductive step: d(m^j ⇝ m^i) > 0

Suppose the lemma holds true for all n^l and k such that d(n^k ⇝ n^l) < d(m^j ⇝ m').
By the definitions of CP and CP-distance, there exist \( n^l \) and \( k \) such that \( \text{acq}(m)^j \prec_{\text{HB}} \text{rel}(n)^k \prec_{\text{CP}} \text{acq}(n)^l \), \( d(m^j \leadsto m^i) = 1 + d(n^k \leadsto n^l) \), and \( n \neq m \). Either \( \text{rel}(m)^i \prec_{tr} \text{rel}(n)^l \) or \( \text{rel}(n)^l \prec_{tr} \text{rel}(m)^i \):

**Case 1:** \( \text{rel}(m)^i \prec_{tr} \text{rel}(n)^l \)

Since \( \text{acq}(m)^j \prec_{\text{HB}} \text{rel}(n)^k \), at \( \text{acq}(n)^l \), there exists \( k' \leq k \) such that Algorithm 3 adds \( T': n^k \) to \( \text{CCP}(m^j) \), where \( T' = \text{thr}(\text{acq}(n)^l) \). Since \( \text{acq}(n)^l \prec_{\text{HB}} \text{rel}(m)^i \), and the algorithms propagate CCP through HB-ordered events, at \( e = \text{rel}(m)^i \), \( T': n^k \in \text{CCP}(m^j) \).

**Case 2:** \( \text{rel}(n)^l \prec_{tr} \text{rel}(m)^i \)

To prove this case, we are interested in the release of a lock like \( n \) with “minimum distance.” Let \( o^f \) and \( g \) be such that

- \( \text{rel}(o)^g \prec_{tr} \text{rel}(m)^l \);
- there exists \( f' \geq f \) and \( \sigma \) such that, at \( \text{rel}(o)^g \),
  \[ \exists j' \geq j \mid \sigma: o^{f'} \in \text{CCP}(m^{j'}) \land \exists e' \mid \text{acq}(m)^j \prec_{\text{CP}} e' \land \text{appl}(\sigma, e') \land e' \prec_{\text{HB}} e; \text{ and} \]
- \( d(o^{f'} \leadsto o^g) \) is as minimal as possible.

Note that \( o \) may be \( n \), or else \( o \) is some other “lower-distance” lock, and thus \( d(o^{f'} \leadsto o^g) < d(m^j \leadsto m^i) \) in any case. It is possible that \( o = m \), in which case \( f < j \) and \( g < i \).

Since \( d(o^{f'} \leadsto o^g) < d(m^j \leadsto m^i) \), by the inductive hypothesis, at \( \text{rel}(o)^g \) (after the pre-release algorithm but before the release algorithm), there exists \( f' \geq f \) such that either

- \( T' \in \text{CP}(o^{f'}) \) or
- \( \exists q^c \mid q \neq o \land T': q^c \in \text{CCP}(o^{f'}) \land \exists d \mid \text{rel}(q)^c \prec_{\text{CP}} \text{acq}(q)^d \prec_{tr} \text{rel}(o)^g \land d(q^c \leadsto q^d) < d(o^{f'} \leadsto o^g) \)
where \( T' = thr(\rel(o)^g) \).

**Case 2a:** \( T' \in CP(o'^f) \)

At \( \rel(o)^g \), the pre-release algorithm (Algorithm 4) adds \( \sigma \) to \( CP(m'^j) \) at line 3 because \( \sigma : o'^f \in CCP(m'^j) \) and \( T' \in CP(o'^f) \) (matching lines 2–3). Since \( \text{appl}(\sigma, e') \land e' \prec_{hb} e \), and the algorithms propagate CP through HB-ordered events, at \( e = \rel(m)^i \), \( T \in CP(m'^j) \).

**Case 2b:** Let \( q^c \) and \( d \) be such that, at \( \rel(o)^g \), \( q \neq o \land T': q^c \in CCP(o'^f) \land \rel(q)^c <_{cp} \text{acq}(q)^d <_{tr} \rel(o)^g \land d(q'^c \leadsto q^d) < d(o^f \leadsto o^g) \).

Thus \( \text{acq}(q)^d <_{tr} \rel(o)^g <_{tr} \rel(q)^d \).

If \( \rel(q)^d <_{tr} \rel(m)^i \), then that would violate the stipulation above that \( d(o^f \leadsto o^g) \) is minimal. Thus either \( \rel(m)^i = \rel(q)^d \) or \( \rel(m)^i <_{tr} \rel(q)^d \).

**Case 2b(i):** \( \rel(m)^i = \rel(q)^d \)

Thus \( m = q \) and \( i = d \), but \( c < j \) since \( d(m'^c \leadsto m'^j) = d(q'^c \leadsto q^d) < d(m'^i \leadsto m'^j) \).

At \( \rel(o)^g \), the pre-release algorithm (Algorithm 4) adds \( \sigma : m'^c \) to \( CCP(m'^j) \) at line 5 because \( \sigma : o'^f \in CCP(m'^j) \) (matching line 2) and \( T' : m'^c \in CCP(o'^f) \) (matching line 4). Since \( \text{appl}(\sigma, e') \land e' \prec_{hb} e \), and the algorithms propagate CCP through HB-ordered events, at \( e = \rel(m)^i \), \( T : m'^c \in CCP(m'^j) \).

By the inductive hypothesis on \( m'^i \) and \( c \), since \( d(m'^c \leadsto m'^j) < d(m'^i \leadsto m'^j) \), there exists \( c' \geq c \) such that, at \( \rel(m)^i \), one or both of the following hold:

- \( T \in CP(m'^c) \)

The pre-release algorithm (Algorithm 4) adds \( T \) to \( CP(m'^j) \) at line 3 because \( T : m'^c \in CCP(m'^j) \) (matching line 2) and \( T \in CP(m'^c) \) (matching line 3).
• $\exists r^a \mid r \neq m \land T : r^a \in CCP(m^{c'}) \land \exists b \mid rel(r)^a <_{CP} acq(r)^b <_{tr} rel(m)^i \land d(r^a \leadsto r^b) < d(m^{c'} \leadsto m^i)$

The pre-release algorithm (Algorithm 4) adds $T : r^a$ to $CCP(m^j')$ at line 5 because $T : m^{c'} \in CCP(m^j')$ (matching line 2) and $T : r^a \in CCP(m^{c'})$ (matching line 4).

**Case 2b(ii):** $rel(m)^i <_{tr} rel(q)^d$

Note that $m \neq q$ because $acq(q)^d <_{tr} rel(o)^g <_{tr} rel(m)^i <_{tr} rel(q)^d$.

At $rel(o)^g$, the pre-release algorithm (Algorithm 4) adds $\sigma : q^c$ to $CCP(m^j')$ at line 5 because $\sigma : o^{f'} \in CCP(m^j')$ (matching line 2) and $T' : q^c \in CCP(o^{f'})$ (matching line 4). Since $appl(\sigma, e') \land e' \prec_{HB} e$, and the algorithms propagate CCP through HB-ordered events, at $e = rel(m)^i$, $T : q^{c'} \in CCP(m^j')$.

Thus, the lemma’s statement holds for each case.

We are ready to prove Theorem 1, which states that after every event, Raptor maintains the invariants in Figure 3.2.

**Proof.** By induction on the observed total order of events ($<_{tr}$).

**Base case:** Let $e$ be a first “no-op” event in $<_{tr}$ that precedes all program events and has no effect on analysis state. Before and thus after $e$, all CP and CCP sets are empty, so the LHS of the [CP] invariant is $\emptyset$. The RHS of the [CP] invariant is $\emptyset$ because there is no earlier event $e' <_{tr} e$.

**Inductive step:** Suppose the invariants in Figure 3.2 hold immediately before an event $e$ executed by thread $T$. Let $e^+$ be the event immediately after $e$ in the observed total order.
Let \( \rho \) be any set owner. Let \( e_\rho \) be the event corresponding to \( \rho \), i.e., \( e_\rho = \text{wr}(x)_T \) if \( \rho = x^h \), \( e_\rho = \text{rd}(x)_T \) if \( \rho = x^h \), or \( e_\rho = \text{acq}(m)^h \) if \( \rho = m^h \).

We define the following abbreviations for the left- and right-hand sides of the \([\text{CP}]\) invariant:

Let \( \text{LHS} = \text{CP}(\rho) \cup \{ \sigma \mid (\exists n^k \mid \sigma : n^k \in \text{CCP}(\rho) \land \exists j \mid \text{rel}(n)^k <_{cp} \text{acq}(n)^j <_{tr} e) \} \).

Let \( \text{LHS}^+ = \text{CP}(\rho)^+ \cup \{ \sigma \mid (\exists n^k \mid \sigma : n^k \in \text{CCP}(\rho)^+ \land \exists j \mid \text{rel}(n)^k <_{cp} \text{acq}(n)^j <_{tr} e^+) \} \).

Let \( \text{RHS} = \{ \sigma \mid (\exists e' \mid \text{appl}(\sigma, e') \land e_\rho <_{cp} e^+ <_{tr} e) \} \).

Let \( \text{RHS}^+ = \{ \sigma \mid (\exists e' \mid \text{appl}(\sigma, e') \land e_\rho <_{cp} e^+ <_{tr} e^+) \} \).

Inductive hypothesis: Suppose Figure 3.2’s invariants hold before \( e \), i.e., \( \text{LHS} = \text{RHS} \).

If \( e \) is a release, we consider the analysis state before \( e \) to be the state after the pre-release algorithm (Algorithm 4) has executed for \( e \), i.e., let \( \text{LHS} \) reflect the state after the pre-release algorithm has executed. By Lemma 1, Figure 3.2’s invariants (i.e., \( \text{LHS} = \text{RHS} \)) hold at this point.

To show \( \text{LHS}^+ = \text{RHS}^+ \), we show \( \text{LHS}^+ \subseteq \text{RHS}^+ \) (subset) and \( \text{LHS}^+ \supseteq \text{RHS}^+ \) (superset) in turn.

**Subset direction:** Let \( \sigma \in \text{LHS}^+ \).

Either \( \sigma \in \text{CP}(\rho)^+ \) or \( \exists n^k \mid \sigma : n^k \in \text{CCP}(\rho)^+ \land \exists j \mid \text{rel}(n)^k <_{cp} \text{acq}(n)^j <_{tr} e^+ \) (or both):

**Case 1:** \( \sigma \in \text{CP}(\rho)^+ \)

If \( \sigma \in \text{CP}(\rho) \), then by the inductive hypothesis, \( \sigma \in \text{RHS} \), i.e., \( \exists e' \mid \text{appl}(\sigma, e') \land e_\rho <_{cp} e^+ <_{tr} e \). Since \( e <_{tr} e^+ \), \( \exists e' \mid e_\rho <_{cp} e^+ <_{tr} e^+ \), so \( \sigma \in \text{RHS}^+ \).

Otherwise (\( \sigma \notin \text{CP}(\rho) \)), the analysis adds \( \sigma \) to \( \text{CP}(\rho)^+ \):
Case 1a: \( e = \text{wr}(x)^j \). Algorithm 1 adds \( \sigma \) to \( CP(\rho)^+ \) at line 2, 4, 7 or 12:

- Lines 2 and 4 execute within line 1’s for loop, so let \( m \in \text{heldBy}(T) \). Since line 2 or 4 executes, the if condition in line 2 or 4 evaluates to true, so let \( h < i \) be such that (1) \( T \notin PO(x^h) \) or \( T \notin PO(x^h_t) \), and (2) let \( j \) be such that \( m_j^i \in HB(x^h) \) or \( m_j^i \in HB(x^h_t) \), respectively. Lines 2 and 4 add \( T \) to \( CP(m_j^i) \), so \( \rho = m_j^i \) and \( \sigma = T \).

By the inductive hypothesis on \( m_j^i \in HB(x^h) \) or \( m_j^i \in HB(x^h_t) \) ([HB-critical-section] invariant), \( \text{acq}(m)^j \prec_{po} e'' \prec_{po} \text{rel}(m)^j \land e'' \prec_{tr} e \), where \( e'' = \text{wr}(x)^h \) or \( \text{rd}(x)^h \). Because \( m \in \text{heldBy}(T) \) and \( T \notin PO(x^h) \) or \( T \notin PO(x^h_t) \), let \( k > j \) be such that \( \text{acq}(m)^k \prec_{po} e \prec_{po} \text{rel}(m)^k \). By CP Rule (a), \( \text{rel}(m)^j \prec_{cp} \text{acq}(m)^k \). By CP Rule (c), \( \text{acq}(m)^j \prec_{cp} e \). Since \( \text{thr}(e) = T \land \text{acq}(m)^j \prec_{cp} e \prec_{tr} e^+ \), therefore \( T \in \text{RHS}^+ \).

- Line 7 adds \( \xi \) to \( CP(\rho)^+ \), so \( \rho = x^{i-1} \), \( e_\rho = \text{wr}(x)^{i-1} \), \( \sigma = \xi \), and \( T \in CP(\rho) \) (line 7). By the inductive hypothesis ([CP] invariant), let \( e' \) be such that \( \text{thr}(e') = T \land \text{wr}(x)^{i-1} \prec_{cp} e' \prec_{tr} e \). By CP Rule (c), since \( \text{thr}(e') = T = \text{thr}(e) \), \( \text{wr}(x)^{i-1} \prec_{cp} e \prec_{tr} e^+ \). Note that \( e = \text{wr}(x)^j \) is \( e_\xi \) from Figure 3.2. Therefore \( \xi \in \text{RHS}^+ \).

- Line 12 adds \( \xi \) to \( CP(\rho)^+ \), so \( \rho = x^j_t \), \( e_\rho = \text{rd}(x)^j_t \), \( \sigma = \xi \), and \( T \in CP(\rho) \) (line 12). By the inductive hypothesis ([CP] invariant), let \( e' \) be such that \( \text{thr}(e') = T \land \text{rd}(x)^j_t \prec_{cp} e' \prec_{tr} e \). By CP Rule (c), since \( \text{thr}(e') = T = \text{thr}(e) \), \( \text{rd}(x)^j_t \prec_{cp} e \prec_{tr} e^+ \). Note that \( e = \text{wr}(x)^j \) is \( e_\xi \) from Figure 3.2. Therefore \( \xi \in \text{RHS}^+ \).

Case 1b: \( e = \text{rd}(x)^j_T \). Algorithm 2 adds \( \sigma \) to \( CP(\rho)^+ \) at line 2 or 5:
• Line 2 executes within line 1’s for loop, so let \( m \in heldBy(T) \). Since line 2 executes, its if condition evaluates to true, so let \( h < i \) be such that (1) \( T \notin PO(x^h) \), and (2) let \( j \) be such that \( m^j_i \in HB(x^h) \). Line 2 adds \( T \) to \( CP(m^j)^+ \), so \( \rho = m^j \) and \( \sigma = T \).

By the inductive hypothesis on \( m^j_i \in HB(x^h) \) ([HB-critical-section] invariant), \( acq(m)^j \prec_{po} wr(x)^h \prec_{po} rel(m)^j \land wr(x)^h <_{tr} e \). Because \( m \in heldBy(T) \) and \( T \notin PO(x^h) \), let \( k > j \) be such that \( acq(m)^k \prec_{po} e \prec_{po} rel(m)^k \). By CP Rule (a), \( rel(m)^j \prec_{cp} acq(m)^k \). By CP Rule (c), \( acq(m)^j \prec_{cp} e <_{cp} e \). Since \( thr(e) = T \land acq(m)^j \prec_{cp} e <_{cp} e \), therefore \( T \in RHS^+ \).

• Line 5 adds \( \xi_T \) to \( CP(\rho)^+ \), so \( \rho = \xi^i \), \( e_\rho = wr(x)^i \), \( \sigma = \xi_T \), and \( T \in CP(\rho) \) (line 5). By the inductive hypothesis ([CP] invariant), let \( e' \) be such that \( thr(e') = T \land wr(x)^i \prec_{cp} e' <_{tr} e \). By CP Rule (c), since \( thr(e') = T = thr(e) \), \( wr(x)^i \prec_{cp} e <_{cp} e <_{tr} e^+ \). Note that \( e = rd(x)^i_T \) is \( e_{\xi_T}^i \) from Figure 3.2. Therefore \( \xi_T \in RHS^+ \).

Case 1c: \( e = acq(m)^j \). Algorithm 3 adds \( \sigma \) to \( CP(\rho)^+ \) only at line 2, which adds \( T \) to \( CP(\rho)^+ \), so \( \sigma = T \) and \( m \in CP(\rho) \) (line 2). By the inductive hypothesis ([CP] invariant), \( \exists j \) \( e_\rho \prec_{cp} rel(m)^j <_{tr} e \). By CP Rule (c), \( e_\rho \prec_{cp} e \) (since \( e = acq(m)^j \)). Because \( thr(e) = T \land e_\rho \prec_{cp} e <_{tr} e^+ \), therefore \( T \in RHS^+ \).

Case 1d: \( e = rel(m)^j \). Algorithm 5 adds \( \sigma \) to \( CP(\rho)^+ \) only at line 2, which adds \( m \) to \( CP(\rho)^+ \), so \( \sigma = m \) and \( T \in CP(\rho) \) (line 2). By the inductive hypothesis ([CP] invariant), let \( e' \) be such that \( thr(e') = T \land e_\rho \prec_{cp} e' <_{tr} e \). By the definition of HB, \( e' \prec_{hb} e \), and thus \( e_\rho \prec_{cp} e <_{cp} e^+ \) by CP Rule (c). Therefore \( m \in RHS^+ \).

Case 2: Let \( n^k \) and \( n^j \) be such that \( \sigma : n^k \in CCP(\rho)^+ \land rel(n)^k \prec_{cp} acq(n)^j <_{tr} e^+ \).
If \( \sigma : n^k \in CCP(\rho) \land \text{rel}(n)^k \prec cp \ acc(n)^i \prec tr e \), then by the inductive hypothesis ([CP] invariant), \( \sigma \in RHS \), i.e., \( \exists e' | \text{appl}(\sigma, e') \land e \prec cp e' < tr e \). Since \( e < tr e^+ \), \( e \prec cp e' < tr e^+ \), and thus \( \sigma \in RHS^+ \).

Otherwise, \( \neg (\sigma : n^k \in CCP(\rho) \land \text{rel}(n)^k \prec cp \ acc(n)^i \prec tr e) \). In fact, we can conclude that \( \sigma : n^k \notin CCP(\rho) \) using the following reasoning. If \( \neg (\text{rel}(n)^k \prec cp \ acc(n)^i \prec tr e) \), then \( acc(n)^i \notin tr e \) (since \( \text{rel}(n)^k \prec cp \ acc(n)^i \)) and thus \( e = acq(n)^i \) (since \( acc(n)^i \prec tr e^+ \) and \( e \) immediately precedes \( e^+ \))—in which case the inductive hypothesis ([CCP-constraint] invariant) ensures \( \sigma : n^k \notin CCP(\rho) \). Thus the analysis adds \( \sigma : n^k \) to \( CCP(\rho)^+ \):

**Case 2a:** \( e = wr(x)^j \). Algorithm 1 adds \( \sigma : n^k \) to \( CCP(\rho)^+ \) at line 8 or 13.

- Line 8 adds \( \xi : n^k \) to \( CCP(\rho)^+ \), so \( \rho = x^{i-1} \), \( e_{\rho} = wr(x)^{i-1} \), \( \sigma = \xi \), and \( T : n^k \in CCP(x^{i-1}) \) (line 8). By the inductive hypothesis ([CP] invariant), let \( e' \) be such that \( thr(e') = T \land wr(x)^{i-1} \prec cp e' < tr e \). By the definitions of HB and CP, since \( thr(e') = T = thr(e), wr(x)^{i-1} \prec cp e < tr e^+ \). Note that \( e = wr(x)^i \) is \( e_\xi \) from Figure 3.2. Therefore \( \xi \in RHS^+ \).

- Line 13 adds \( \xi : n^k \) to \( CCP(\rho)^+ \), so \( \rho = x^{i-1} \), \( e_{\rho} = rd(x)^{i-1} \), \( \sigma = \xi \), and \( T : n^k \in CCP(x^{i-1}) \) (line 13). By the inductive hypothesis ([CP] invariant), let \( e' \) be such that \( thr(e') = T \land rd(x)^{i-1} \prec cp e' < tr e \). By the definitions of HB and CP, since \( thr(e') = T = thr(e), rd(x)^{i-1} \prec cp e < tr e^+ \). Note that \( e = wr(x)^i \) is \( e_\xi \) from Figure 3.2 Therefore \( \xi \in RHS^+ \).

**Case 2b:** \( e = rd(x)^i \). Algorithm 2 adds \( \sigma : n^k \) to \( CCP(\rho)^+ \) at line 6. So \( \rho = x^i \), \( e_{\rho} = wr(x)^i \), \( \sigma = \xi_T \), and \( T : n^k \in CCP(x^i) \) (line 6). By the inductive hypothesis ([CP] invariant), let \( e' \) be such that \( thr(e') = T \land wr(x)^i \prec cp e' < tr e \). By the definitions of
HB and CP, since \( \text{thr}(e') = T = \text{thr}(e) \), \( \text{wr}(x)^j <_{\text{cp}} e <_{tr} e^+ \). Note that \( e = \text{rd}(x)^j_T \) is \( e^j_T \) from Figure 3.2. Therefore \( \xi_T \in \text{RHS}^+ \).

**Case 2c:** \( e = \text{acq}(m)^j \). Algorithm 3 adds \( \sigma : n^k \) to \( \text{CCP}(\rho)^+ \) at line 3 or 6.

- If line 3 adds \( \sigma : n^k \) to \( \text{CCP}(\rho)^+ \), then \( \sigma = T \) and \( m : n^k \in \text{CCP}(\rho) \) (line 3). By the inductive hypothesis ([CP] invariant), \( \exists l \mid e^l_\rho <_{\text{cp}} \text{rel}(m)^j <_{tr} e \). By CP Rule (c), \( e^l_\rho <_{\text{cp}} e \) since \( e = \text{acq}(m)^j \). Since \( \text{thr}(e) = T \land e^l_\rho <_{\text{cp}} e <_{tr} e^+ \), therefore \( T \in \text{RHS}^+ \).

- If line 6 adds \( \sigma : n^k \) to \( \text{CCP}(\rho)^+ \), then \( n^k \in \text{HB}(\rho) \) (recall that \( n^k \in \text{HB}(\rho) \) if \( n^k \in \text{HB}(\rho) \)) according to line 4. Furthermore, \( n = m \) and \( \sigma = T \). By the inductive hypothesis ([HB], [HB-index], and [HB-critical-section] invariants), \( e^l_\rho <_{\text{hb}} \text{rel}(n)^k <_{tr} e \). Since we know \( \text{rel}(n)^k <_{\text{cp}} \text{acq}(n)^j <_{tr} e^+ \) (from the beginning of Case 2) and \( e = \text{acq}(n)^j \) (since \( n = m \)), therefore \( e^l_\rho <_{\text{cp}} e \) by CP Rule (c) (regardless of whether \( j = i \) or \( j < i \)). Since \( \text{thr}(e) = T \land e^l_\rho <_{\text{cp}} e <_{tr} e^+ \), \( T \in \text{RHS}^+ \).

**Case 2d:** \( e = \text{rel}(m)^j \). Algorithm 5 adds \( \sigma : n^k \) to \( \text{CCP}(\rho)^+ \) only at line 3, so \( \sigma = m \) and \( T : n^k \in \text{CCP}(\rho) \) (line 3). By the inductive hypothesis ([CP] invariant), \( \exists l \mid e^l_\rho <_{\text{cp}} \text{rel}(m)^j <_{tr} e \) since \( \sigma = m \). Since \( \text{rel}(m)^j <_{\text{hb}} \text{acq}(m)^j \) and \( e = \text{rel}(m)^j \) and \( \text{thr}(e) = T \), by CP Rule (c), \( e^l_\rho <_{\text{cp}} e <_{tr} e^+ \). Therefore \( m \in \text{RHS}^+ \).

**Superset direction:** Let \( \sigma \in \text{RHS}^+ \). Either \( \sigma \in \text{RHS} \) or not:

**Case 1:** \( \sigma \in \text{RHS} \)

Then \( \sigma \in \text{LHS} \) by the inductive hypothesis. If \( e \) is a write, read, or acquire, then \( \sigma \in \text{LHS}^+ \) because Algorithms 1, 2, and 3 maintain \( \text{CP}(\rho)^+ \supseteq \text{CP}(\rho) \) and \( \text{CCP}(\rho)^+ \supseteq \text{CCP}(\rho) \)
CCP(\(\rho\)), \(\sigma \in LHS^+\). (Line 9 of Algorithm 2 overwrites \(CP(x^T_i)^+\) and \(CCP(x^T_i)^+\), which is correct because, in essence, the latest \(rd(x^T_i)\) “replaces” any previous \(rd(x^T_i)\). Thus for \(\rho = x^T_i\) or \(\rho = x^T_i\), no \(\sigma \in RHS\) exists.) But we must consider \(e = rel(m)^j\) since Algorithm 5 removes some CCP elements. Since \(\sigma \in LHS\), \(\sigma \in CP(\rho)\) or \(\exists n^k | \sigma : n^k \in CCP(\rho) \land \exists j | rel(n)^k \prec_{cp} acq(n)^j <_{tr} e\):

**Case 1a:** If \(\sigma \in CP(\rho)\), then \(\sigma \in CP(\rho)^+\) because Algorithm 5 maintains \(CP(\rho)^+ \supseteq CP(\rho)\).

**Case 1b:** Let \(n^k\) be such that \(\sigma : n^k \in CCP(\rho) \land \exists j | rel(n)^k \prec_{cp} acq(n)^j <_{tr} e\).

If \(n \neq m\), then Algorithm 5 does not remove \(\sigma : n^k\) from \(CCP(\rho)^+\), so \(\sigma : n^k \in CCP(\rho)^+\).

Otherwise, \(n = m\). Since \(\exists j | rel(n)^k \prec_{cp} acq(n)^j <_{tr} e\) and \(e = rel(m)^j\), therefore \(acq(m)^k \prec_{cp} acq(m)^j\) by CP Rule (c) and Rule (b). By the inductive hypothesis ([CP] invariant) and Lemma 2, \(T \in CP(m^k)\) or \(\exists o^l | o \neq m \land T : o^l \in CCP(m^k) \land \exists h | rel(o)^l \prec_{cp} rel(o)^h <_{tr} e\).

**Case 1b(i):** \(T \in CP(m^k)\)

In the pre-release algorithm (Algorithm 4), line 3 evaluated to true for \(\sigma : m^l \in CCP(\rho) | l \leq k\) (line 2). Thus line 3 executed, so \(\sigma \in CP(\rho)\). Since the release algorithm (Algorithm 5) maintains \(CP(\rho)^+ \supseteq CP(\rho)\), therefore \(\sigma \in LHS^+\).

**Case 1b(ii):** Let \(o^l\) be such that \(o \neq m \land T : o^l \in CCP(m^k) \land \exists h | rel(o)^l \prec_{cp} rel(o)^h <_{tr} e\).

Then the pre-release algorithm (Algorithm 4) executed line 5 for \(\sigma : m^k \in CCP(\rho)\) (line 2) and \(T : o^l \in CCP(m^k)\) (line 4). Line 5 added \(\sigma : o^l\) to \(CCP(\rho)\).
The release algorithm (Algorithm 5) does not remove $\sigma : o_\downarrow$ because $o \neq m$ (line 5).

Therefore $\sigma : o_\downarrow \in CCP(\rho)^+$ and thus $\sigma \in LHS^+$.

Case 2: $\sigma \notin RHS$

Thus $\exists e' | appl(\sigma, e') \land e_\rho \prec_{cp} e' <_{tr} e^+$ but $\nexists e' | appl(\sigma, e') \land e_\rho \prec_{cp} e' <_{tr} e$, which implies $appl(\sigma, e) \land e_\rho \prec_{cp} e$. $e$ is either a write, read, acquire, or release:

Case 2a: $e = wr(x)^i$

By the definition of CP, since $e$ is not an acquire, $\exists e | thr(e') = T \land e_\rho \prec_{cp} e' <_{tr} e$, i.e., $T \in RHS$. According to the definition of $appl(\sigma, e)$, either (1) $\sigma = T$ or (2) $\sigma = \xi$ (note that $e = wr(x)^i$ is $e_\xi$ from Figure 3.2). However, $\sigma \notin RHS$, so $\sigma = \xi$.

Therefore $\rho = x^{i-1}$, $e_\rho = wr(x)^{i-1}$, and $wr(x)^{i-1} \prec_{cp} e$; or $\rho = x^{i-1}_T$, $e_\rho = rd(x)^{i-1}_T$, and $rd(x)^{i-1}_T \prec_{cp} e$. By the inductive hypothesis on $T \in RHS (\{CP\}$ invariant), $T \in CP(x^{i-1})$, $T \in CP(x^{i-1}_t)$, $\exists n^k | T : n^k \in CCP(x^{i-1}) \land \exists j \land rel(n)^k \prec_{cp} acq(n)^j <_{tr} e$, or $\exists n^k | T : n^k \in CCP(x^{i-1}_t) \land \exists j \land rel(n)^k \prec_{cp} acq(n)^j <_{tr} e$.

Case 2a(i): $T \in CP(x^{i-1})$

Then Algorithm 1’s line 7 evaluates to true, so line 7 adds $\xi$ to $CP(x^{i-1})^+$; thus $\xi \in LHS^+$.

Case 2a(ii): $T \in CP(x^{i-1}_t)$

Line 12 evaluates to true, so line 12 adds $\xi$ to $CP(x^{i-1}_t)$; thus $\xi \in LHS^+$.

Case 2a(iii): Let $n^k$ be such that $T : n^k \in CCP(x^{i-1}) \land \exists j \land rel(n)^k \prec_{cp} acq(n)^j <_{tr} e$.

Since $T : n^k \in CCP(x^{i-1})$ matches line 8 in Algorithm 1, line 8 adds $\xi : n^j$ to $CCP(x^{i-1})^+$. So $\exists j \land rel(n)^k \prec_{cp} acq(n)^j <_{tr} e <_{tr} e^+$; thus $\xi \in LHS^+$.

Case 2a(iv): Let $n^k$ be such that $T : n^k \in CCP(x^{i-1}_t) \land \exists j \land rel(n)^k \prec_{cp} acq(n)^j <_{tr} e$. 

Since $T : n^k \in CCP(x_{t-1}^i)$ matches line 13 in Algorithm 1, line 13 adds $\xi : n^j$ to $CCP(x_{t-1}^i)$. So $\exists j \mid rel(n)^k \prec_{cp} acq(n)^j < tr e < tr e^+$; thus $\xi \in LHS^+$.

**Case 2b:** $e = rd(x)^l_T$

By the definition of CP, since $e$ is not an acquire, $\exists e' \mid thr(e') = T \land e_\rho \prec_{cp} e' < tr e$, i.e., $T \in RHS^+$. According to the definition of $appl(\sigma, e)$, either (1) $\sigma = T$ or (2) $\sigma = \xi_T$ (note that $e = rd(x)^l_T$ is $e_\xi_T$ from Figure 3.2). However, $\sigma \not\in RHS$, so $\sigma = \xi_T$. Therefore $\rho = x^l$, $e_\rho = wr(x)^l$, and $wr(x)^l \prec_{cp} e$. By the inductive hypothesis on $T \in RHS$ ([CP] invariant), $T \in CP(x^l)$ or $\exists n^k \mid T : n^k \in CCP(x^l) \land \exists j \mid rel(n)^k \prec_{cp} acq(n)^j < tr e$.

**Case 2b(i):** $T \in CP(x^l)$

Then Algorithm 2’s line 5 evaluates to true, so line 5 adds $\xi_T$ to $CP(x^l)^+$; thus $\xi_T \in LHS^+$.

**Case 2b(ii):** Let $n^k$ be such that $T : n^k \in CCP(x^l) \land \exists j \mid rel(n)^k \prec_{cp} acq(n)^j < tr e$.

Since $T : n^k \in CCP(x^l)$ matches line 6 in Algorithm 2, line 6 adds $\xi_T : n^j$ to $CCP(x^l)$. So $\exists j \mid rel(n)^k \prec_{cp} acq(n)^j < tr e < tr e^+$; thus $\xi_T \in LHS^+$.

**Case 2c:** $e = acq(m)^l$

Thus $\sigma = T$. Since $e_\rho \prec_{cp} e$ and $\exists e' \mid thr(e') = T \land e_\rho \prec_{cp} e' < tr e$ (recall $\sigma \not\in RHS$), by CP Rule (c), $\exists l \mid e_\rho \prec_{cp} rel(m)^l <_{hb} e$ or $\exists l \mid e_\rho \prec_{hb} rel(m)^l \prec_{cp} e$.

**Case 2c(i):** $\exists l \mid e_\rho \prec_{cp} rel(m)^l <_{hb} e$

By the inductive hypothesis on $e_\rho \prec_{cp} rel(m)^l$ ([CP] invariant), $m \in CP(\rho)$ or $\exists n^k \mid m : n^k \in CCP(\rho) \land \exists j \mid rel(n)^k \prec_{cp} acq(n)^j < tr e$.

- $m \in CP(\rho)$
Then Algorithm 3’s line 2 evaluates to true, so line 2 adds $T$ to $CP(\rho)^+$; thus $T \in LHS^+$.

- Let $n^k$ be such that $m : n^k \in CCP(\rho) \land \exists j \mid rel(n)^k \prec_{cp} acq(n)^j < tr_e$.

  Then $m : n^k \in CCP(\rho)$ matches line 3 in Algorithm 3, so line 3 adds $T : n^k$ to $CCP(\rho)^+$. So $\exists j \mid rel(n)^k \prec_{cp} acq(n)^j < tr_e < tr_e + e^+$; thus $T \in LHS^+$.

Case 2c(ii): $\exists l \mid e_{\rho} \prec_{HB} rel(m)^l \prec_{HB} CP_e$.

Let $j$ be the minimum value such that $e_{\rho} \prec_{HB} rel(m)^j$. By the inductive hypothesis (\{HB\}, \{HB-index\}, and \{HB-critical-section\} invariants), $m^j \in HB(\rho)$ (recall that $m^j \in HB(\rho)$ if $m^i \in HB(\rho)$). Thus line 4 evaluates to true for $\rho$, so line 6 executes and $T : m^j \in CCP(\rho)^+$. Furthermore, $rel(m)^j \prec_{cp} acq(m)^j < tr_e + (e = acq(m)^j)$. Therefore $T \in LHS^+$.

Case 2d: $e = rel(m)^j$.

Since $e$ is not an acquire, by CP Rule (c), $\exists e' \mid thr(e') = T \land e_{\rho} \prec_{cp} e' < tr_e$, i.e., $T \in RHS$. According to the definition of $appl(\sigma, e)$, either $\sigma = T$ or $\sigma = m$. But $\sigma \neq T$ since $\sigma \notin RHS$, so $\sigma = m$. By the inductive hypothesis (\{CP\} invariant) on $T \in RHS$, $T \in CP(\rho)$ or $\exists n^k \mid T : n^k \in CCP(\rho) \land \exists j \mid rel(n)^k \prec_{cp} acq(n)^j < tr_e$.

Case 2d(i): $T \in CP(\rho)$.

Then Algorithm 5’s line 2 evaluates to true, so line 2 adds $m$ to $CP(\rho)^+$; thus $m \in LHS^+$.

Case 2d(ii): Let $n^k$ and $n^j$ be such that $T : n^k \in CCP(\rho) \land rel(n)^k \prec_{cp} acq(n)^j < tr_e$.

If $n \neq m$, then $T : n^k \in CCP(\rho)$ matches line 3 in Algorithm 5, so line 3 adds $m : n^k$ to $CCP(\rho)^+$. Since $e < tr_e + e^+$, $m : n^k \in CCP(\rho) \land rel(n)^k \prec_{cp} acq(n)^j < tr_e + e^+$, i.e., $m \in LHS^+$.
Otherwise \((n = m)\), \(m : m^k \notin CCP(\rho)^+\) because Algorithm 5 removes \(m : m^k\) from \(CCP(\rho)^+\) at line 5. Since \(rel(m)^k \prec_{CP} acq(m)^j \prec_{tr} e\), \(acq(m)^k \prec_{CP} acq(m)^j \prec_{tr} e\) by CP Rule (c) (since \(e = rel(m)^i\)). By the inductive hypothesis ([CP] invariant) and Lemma 2, \(T \in CP(m^k)\) or \(\exists o^l \mid o \neq m \land T : o^l \in CCP(m^k) \land \exists h \mid rel(o)^l \prec_{CP} acq(o)^h \prec_{tr} e\).

- \(T \in CP(m^k)\)

  In the pre-release algorithm (Algorithm 4), line 3 evaluated to true for \(T : m^k \in CP(\rho)\) (line 2). Thus line 3 executed, so \(T \in CP(\rho)\). In the release algorithm (Algorithm 5), line 2 evaluates to true for \(\rho\), so \(m \in CP(\rho)^+\) and thus \(m \in LHS^+\).

- Let \(o^l\) and \(o^h\) be such that \(T : o^l \in CCP(m^k) \land rel(o)^l \prec_{CP} acq(o)^h \prec_{tr} e\).

  In the pre-release algorithm (Algorithm 4), line 5 executed for \(T : m^k \in CCP(\rho)\) (line 2) and \(T : o^l \in CCP(m^k)\) (line 4). Line 5 added \(T : o^l\) to \(CCP(\rho)\). In the release algorithm (Algorithm 5), line 3 executes for \(T : o^l \in CCP(\rho)\) (line 3), adding \(m : o^l\) to \(CCP(\rho)^+\). In addition, \(rel(o)^l \prec_{CP} acq(o)^h \prec_{tr} e^+\) since \(e \prec_{tr} e^+\). Therefore \(m \in LHS^+\).

Since \(LHS^+ \subseteq RHS^+\) and \(LHS^+ \supseteq RHS^+, LHS^+ = RHS^+.\)

### 3.8 Evaluation

This section evaluates the performance and CP-race coverage of an implementation of Raptor.
3.8.1 Implementation

Our implementation of the Raptor analysis is built on *RoadRunner* version 0.3 as discussed in Section 2.4 (page 20). Our implementation of Raptor is publicly available.\(^9\)

**Handling non-lock synchronization.** The Raptor analysis handles variable read and write events and lock acquire and release events as depicted in Algorithms 1–7. As mentioned in Section 2.4, Raptor analysis supports additional synchronization primitives by translating each synchronization event to a critical section surrounding the access on a lock unique to the variable at run time. The translation conservatively orders events according to Rule (a) of Definition 10.

Smaragdakis et al. translate these synchronization operations similarly before feeding them to their Datalog CP implementation [92].

**Removing obsolete sets and reporting CP-races.** The implementation follows the logic from Algorithms 6 and 7 to remove obsolete set owners and report CP-races. However, instead of executing these algorithms directly (e.g., periodically passing over all non-obsolete sets and explicitly clearing obsolete set owners), the implementation performs *reference counting* to identify CP-races and remove obsolete sets. The implementation tracks the numbers of remaining \(\xi : m^i\) and \(\xi_T : m^i\) elements in \(CP(x^i)\) (or \(CCP(x^i_T)\)). If any of these counts drop to zero, and the expected \(\xi\) or \(\xi_T\) element(s) are not in \(CP(x^i)\) (or \(CP(x^i_T)\)), then the implementation reports a CP-race. After reporting the race, as described in Section 2.4, the implementation adds the corresponding \(\xi\) or \(\xi_T\) element to \(CP(x^i)\) (or \(CP(x^i_T)\)), effectively simulating a CP-race-free execution up to the current event and avoiding reporting false CP-races downstream; Datalog CP behaves similarly [92]. If all

\(^9\)https://github.com/PLaSSticity/Raptor.git
counts drop to zero, the implementation concludes that any races between \( wr(x)_i \) (or \( rd(x)_i^T \)) and a following access have already been detected or cannot occur, and thus its set owner \( x_i \) (or \( x_i^T \)) is obsolete.

The implementation removes an obsolete set owner (Algorithms 6 and 7) by removing all strong references to it, allowing it to be garbage collected [37]. Variable owners are kept alive by the locks corresponding to \( \xi : m^j \) and \( \xi_T : m^j \) elements in \( CCP(x_i) \) (or \( CCP(x_i^T) \)) referencing the variable. Lock owners are kept alive by variables containing \( \xi : m^j \) and \( \xi_T : m^j \) elements in \( CCP(x_i) \) (or \( CCP(x_i^T) \)). At a release event, the implementation removes \( \xi : m^j \) and \( \xi_T : m^j \) elements from lock and variable owners along with the lock’s references to variables.

**Optimization.** Our prototype implementation of Raptor is largely unoptimized. We have however implemented the following optimization out of necessity. Before the pre-release algorithm (Algorithm 4) iterates over all active set owners \( \rho \), it pre-computes the following information: (1) the maximum \( j \) such that \( \exists l \geq j \mid T \in CP(m^l) \); and (2) for each \( j \), the set of \( n^k \) such that \( \exists l \geq j \mid T : n^k \in CCP(m^l) \). This pre-computation enables quick lookups at lines 3–5 of Algorithm 4, significantly outperforming an unoptimized pre-release algorithm.

**3.8.2 Methodology**

We evaluate Raptor as discussed in Section 2.5 (page 22) on two sets of benchmarks,

- Benchmarks evaluated by Smaragdakis et al. [92] that we were able to obtain and run. Of their benchmarks that we do not evaluate here, all except StaticBucketMap execute fewer than 1,000 events [92].

- The DaCapo benchmarks as discussed in Section 2.5 (page 22).
Datalog CP implementation. We have extended our Raptor implementation to generate a trace of events in a format that can be processed by Smaragdakis et al.’s Datalog CP implementation [92], which they have shared with us. The Raptor and Datalog CP implementations thus analyze identical executions. Our experiments run the Datalog CP implementation with a bounded window size of 500, 5,000, and 50,000 events; Smaragdakis et al. used a window size of 500 events [92].

FastTrack implementation. RoadRunner implements the state-of-the-art FastTrack HB analysis [31]. We added counting of events to the FastTrack implementation in order to report event counts, which also makes FastTrack’s performance more directly comparable with Raptor’s (which also counts events).

3.8.3 Coverage and Performance

This section focuses on answering two key empirical questions. (1) For real program executions, how many CP-races does Raptor detect that HB detectors cannot detect? (2) For real program executions, how many CP-races does Raptor detect that the Datalog CP implementation cannot detect due to its bounded analysis window?

Table 3.1 reports execution time and HB- and CP-race coverage of Raptor, compared with FastTrack’s time and Datalog CP’s time and CP-race coverage for various event window sizes. The Events columns report the number of events (writes, reads, acquires, and releases) processed by FastTrack and Raptor (including additional events generated by translation; Section 3.8.1). FastTrack reports significantly different event counts than Raptor because, for some of the programs, Raptor is limited to analyzing only events within 2 hours. FTPServer is an exception; despite running to completion, Raptor reports fewer events than...
### Table 3.1: Dynamic execution characteristics and detected HB- and CP-races reported for the evaluated programs.

*Events* is the number of program events processed by FastTrack and Raptor, and *#Thr* is the execution’s thread count. *Time* is wall clock execution time, reported in seconds or hours and rounded to two significant figures or the nearest integer if $\geq 100$. For each race column (*HB&CP, HB, CP*), the first number is distinct static races, and the second number (in parentheses) is dynamic races. Each run of FastTrack, Raptor, and Datalog CP is limited to 2 hours; *N/A* columns indicate that Datalog CP did not finish in 2 hours and thus did not report any CP-races.

<table>
<thead>
<tr>
<th></th>
<th>FastTrack [31]</th>
<th>Raptor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#Thr Events Time</td>
<td>#Thr Events Time</td>
</tr>
<tr>
<td>elevator</td>
<td>3 29K 25 s</td>
<td>3 30K 28 s</td>
</tr>
<tr>
<td>FTPServer</td>
<td>12 743K 55 s</td>
<td>13 63K 358 s</td>
</tr>
<tr>
<td>hedc</td>
<td>7 7K 4.7 s</td>
<td>7 8K 2.5 s</td>
</tr>
<tr>
<td>Jigsaw</td>
<td>70 1,774K 37 s</td>
<td>69 361K 2.0 h</td>
</tr>
<tr>
<td>philo</td>
<td>6 &lt;1K 4.3 s</td>
<td>6 &lt;1K 3.1 s</td>
</tr>
<tr>
<td>tsp</td>
<td>9 923,477K 11 s</td>
<td>9 164K 2.0 h</td>
</tr>
<tr>
<td>avrora</td>
<td>4 694,582K 22 s</td>
<td>2 311K 2.0 h</td>
</tr>
<tr>
<td>eclipse</td>
<td>18 13,131K 29 s</td>
<td>4 321K 2.0 h</td>
</tr>
<tr>
<td>jython</td>
<td>2 79,042K 37 s</td>
<td>2 413K 2.0 h</td>
</tr>
<tr>
<td>pmd</td>
<td>2 3,681K 12 s</td>
<td>2 1,805K 2.0 h</td>
</tr>
<tr>
<td>tomcat</td>
<td>1 81K 3.0 s</td>
<td>1 87K 65 s</td>
</tr>
<tr>
<td>xalan</td>
<td>8 112,753K 16 s</td>
<td>8 2,639K 2.0 h</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Datalog CP [92]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>w = 500</td>
<td>w = 5,000</td>
</tr>
<tr>
<td></td>
<td>Time CP</td>
<td>Time CP</td>
</tr>
<tr>
<td>elevator</td>
<td>38 s</td>
<td>0</td>
</tr>
<tr>
<td>FTPServer</td>
<td>117 s</td>
<td>6 (27)</td>
</tr>
<tr>
<td>hedc</td>
<td>31 s</td>
<td>0</td>
</tr>
<tr>
<td>Jigsaw</td>
<td>294 s</td>
<td>1 (1)</td>
</tr>
<tr>
<td>philo</td>
<td>15 s</td>
<td>0</td>
</tr>
<tr>
<td>tsp</td>
<td>462 s</td>
<td>1 (2)</td>
</tr>
<tr>
<td>avrora</td>
<td>161 s</td>
<td>0</td>
</tr>
<tr>
<td>eclipse</td>
<td>392 s</td>
<td>0</td>
</tr>
<tr>
<td>jython</td>
<td>278 s</td>
<td>0</td>
</tr>
<tr>
<td>pmd</td>
<td>188 s</td>
<td>0</td>
</tr>
<tr>
<td>tomcat</td>
<td>42 s</td>
<td>0</td>
</tr>
<tr>
<td>xalan</td>
<td>0.28 h</td>
<td>5 (11)</td>
</tr>
</tbody>
</table>
FastTrack, we believe, because the program executes significantly different numbers of events depending on the execution’s timing.

Timing-based nondeterminism also leads to different threads counts for FastTrack and Raptor on FTPServer and Jigsaw. avrora and eclipse spawn fewer threads in executions with Raptor than with FastTrack because the Raptor executions do not make enough progress in 2 hours to have spawned all threads.

In our experiments, the Datalog CP implementation processes the same execution trace as Raptor, by processing traces generated by Raptor during its online analysis. Raptor reports significantly different event counts for some of the programs also evaluated by Smaragdakis et al. [92]; this discrepancy makes sense because, at least in several cases, we are using different workloads than prior work.

**Run-time performance.** For FastTrack, the *Time* column reports execution time for optimized HB analysis. For Raptor, the *Time* column reports execution time for Raptor to perform its analysis (which dominates execution time) while also generating a trace of events processed by the Datalog CP implementation. We execute Raptor on each program until it either terminates normally or has executed for two hours. For Datalog CP, the *Time* column reports the cost of executing Datalog CP on the trace generated by the same Raptor execution.

**Race coverage.** The *HB&CP* column reports all races detected by Raptor; the Raptor implementation reports all CP-races and identifies whether it is also an HB-race. The *HB* column reports HB-races (which are always also CP-races), and the *CP* column reports *CP-only* races, which are CP-races that are not also HB-races. For each race column, the first number is *distinct static races*; a static race is an unordered pair of static source locations.
(each source location is a source method and bytecode index). The second number (in parentheses) is dynamic races reported.

Datalog CP, which reports CP-only races by filtering out CP-races that are also HB-races, reports fewer CP-only races than Raptor because of its bounded event window sizes of 500, 5,000, and 50,000 events. Datalog CP does not report any CP-only races if we terminate it after 2 hours (N/A). With a larger window size (5,000 or 50,000 events), Datalog CP sometimes finds more races than at smaller window sizes, but it often cannot complete within 2 hours.

By comparing dynamic event identifiers reported by Datalog CP with those reported by Raptor, we have verified that (1) Raptor detects all CP-only races detected by Datalog CP, and (2) Datalog CP detects each CP-only race reported by Raptor that we would expect it to find (i.e., CP-only races separated by fewer than 500, 5,000, or 50,000 events).

Note that for FTPServer with a 5,000-event window, Datalog CP reports more dynamic CP-only races than Raptor. This phenomenon occurs because Datalog CP reports some races that Raptor filters out by ordering accesses it detects as racing (Section 3.8.1), e.g., if \( \text{wr}(x)^i \) CP-races with \( \text{wr}(x)^{i+2} \), Raptor will report the \( \text{wr}(x)^i - \text{wr}(x)^{i+1} \) and \( \text{wr}(x)^{i+1} - \text{wr}(x)^{i+2} \) CP-races, but Datalog CP will also report the \( \text{wr}(x)^i - \text{wr}(x)^{i+2} \) race. Since Datalog CP reports only dynamic event numbers for CP-races, we have not been able to determine whether the extra 45 dynamic CP-only races reported by Datalog CP represent any additional distinct static CP-only races beyond the 22 distinct static CP-only races reported by Raptor. As a further sanity check of Raptor’s results, we have cross-referenced Raptor’s reported HB-races with results from running RoadRunner’s FastTrack implementation [31] on the same workloads.
Event distance range for each static CP-race

| FTPServer  | 8–1,570; 203–1,472; 253–1,426; 293–1,836; 349–852; 459–2,308; 463–2,309; 495–1,586; 499–2,175; 525–2,415; 510–1,601; 607–1,959; 613–2,505; 760–1,931; 807–2,921; 968–968; 1,023–1,929; 1,397–1,462; 1,453–1,783; 1,736–1,736; 2,063–2,379; 2,079–2,079 |
| hdec       | 2,067–3,774 |
| tsp        | 177–14,303; 939–939 |
| jython     | 157,736–157,739 |
| xalan      | 4–16; 15–32; 15–48; 18–52; 134–149 |

Table 3.2: Event distances for detected CP-races. For each static CP-only race reported by Raptor, the table reports a range of event distances for all dynamic occurrences of the static CP-only race.

**Race characteristics.** Table 3.2 shows the “event distance” for the CP-only races detected by Raptor. The event distance of a CP-race is the distance in the observed total order of events between the race’s two accesses. For each static race in Raptor’s CP column from Table 3.1, Table 3.2 reports the range of event distances for all dynamic instances of that static race. The table shows that many detected dynamic CP-only races have event distances over 500 events and sometimes over 5,000 and 50,000 events; Raptor finds these dynamic CP-only races, but Datalog CP often does not, either due to window size or timeout. For some static CP-only races, every dynamic occurrence has an event distance exceeding 500, 5,000, or 50,000 events, so Datalog CP does not detect the static CP-only race at all, corresponding to CP-only races missed by Datalog CP in Table 3.1. (A few dynamic CP-only races have event distances of just less than 500 events, but Datalog CP does not find them with a window size of 500 events because additional events outside of the window are required to determine CP order.)

In summary, Raptor handles execution traces of 100,000s–1,000,000s of events within two hours, and it finds CP-races that the prior CP analysis cannot find.
3.9 Contribution and Impact

Raptor is concurrent work with *weak-causally-precedes (WCP)* by Kini et al. [47]. Both techniques advance predictive analysis and achieve scalable analysis of long executions. Raptor is the first analysis that computes the causally-precedes (CP) relation online. We have proved that CP maintains invariants needed to track CP soundly and completely. Our evaluation shows that Raptor can analyze execution traces of 100,000s–1,000,000s of events within two hours, allowing it to find CP-races that the prior CP analysis cannot find.

At the time of developing Raptor, all existing predictive analyses could not scale to full program executions, and the CP relation remained the only predictive relation. During this time, we found Raptor was the first predictive analysis to scale to long executions and thus detect races that all prior work was unable to detect. Currently, the WCP analysis can detect more races than any CP analysis in linear time (in trace length). Though WCP is a strict improvement over CP analysis, Raptor remains a useful contribution. Raptor provides a novel lockset-based online analysis for an inherently recursive relation, and unlike prior variants of lockset algorithms, Raptor can soundly detect data races in executions other than the current execution.
Chapter 4: Doesn’t-Commute and Weak-Doesn’t-Commute
Predictive Analysis

4.1 Introduction

Existing partial-order-based predictive analyses weaken the observed ordering more than the happens-before relation while enforcing the same program behavior as the observed execution but miss data races that are knowable from the observed execution. Conservative ordering limits the race coverage of existing predictive analyses by composing with the happens-before relation to provide a sound (detect only true data races) analysis. The composition with happens-before relation inherently leads to poor performance of predictive analyses compared to happens-before analysis. To strengthen race detection capability and improve run time and memory usage, we weaken the ordering between events more than existing partial-order-based predictive analyses while ensuring the same program behavior as the observed execution.

To realize the goal of developing a practical high-coverage sound predictive analysis we introduce two novel partial relations, doesn’t-commute (DC) and weak-doesn’t-commute (WDC), that sacrifice soundness (may detect false races) to detect all data races knowable from the observed execution. The DC and WDC analyses, that track the DC and WDC
relations, expand data race detection capability beyond existing predictive approaches by removing conservative ordering between events.

The DC and WDC analyses reason about weakening the observed order of executing events to improve race coverage beyond existing predictive analyses. Both the DC and WDC relation weaken the WCP relation by avoiding composition with the happens-before relation. The WDC relation further weakens the DC relation by removing tracking of incidental ordering between critical sections. DC and WDC are unsound analyses that achieve high-coverage predictive analysis detecting all data races knowable from the observed execution.

4.2 Doesn’t-Commute Relation

*Doesn’t-commute (DC)* is a new relation that is unsound and high-coverage, detecting all predictable races but also potential false races. Although DC is unsound, it detects few, if any, false races in practice.

**Definition 11** (Doesn’t-commute). Given a trace $tr$, $\prec_{dc}$ is the smallest relation such that:

(a) If two critical sections on the same lock contain conflicting events, then the first critical section is ordered by DC to the second conflicting event. That is, $r_1 \prec_{dc} e_2$ if $r_1$ and $r_2$ are release events on the same lock, $r_1 \prec_{tr} r_2$, $e_1 \in CS(r_1)$, $e_2 \in CS(r_2)$, and $e_1 \approx e_2$.

(b) Release events on the same lock are ordered by DC if their critical sections contain DC-ordered events. Because of the next two rules, this rule can be expressed simply as follows: $r_1 \prec_{dc} r_2$ if $r_1$ and $r_2$ are release events on the same lock and $A(r_1) \prec_{dc} r_2$.

(c) Two events are ordered by DC if they are ordered by PO. That is, $e \prec_{dc} e'$ if $e \prec_{po} e'$.

(d) DC is transitively closed. That is, $e \prec_{dc} e'$ if $\exists e'' \mid e \prec_{dc} e'' \land e'' \prec_{dc} e'$.  

81
Note that DC’s rules (a) and (b) are identical to WCP’s rules (a) and (b), but with $\prec_w$ replaced by $\prec_{wdc}$. DC differs from WCP by composing only with PO, not HB.

An execution trace has a **DC-race** if it has two conflicting events that are unordered by DC. The strict partial order $\prec_{dc}$ is strictly weaker than $\prec_{wcp} \cup \prec_{po}$, and as a result, DC predicts races that WCP does not.

**Examples.** The executions in Figures 2.2, 2.3(a) and 2.4(a) contain DC-races where all “WCP” labeled edges are equivalent to “DC” edges.

Figure 4.1(a) illustrates an example execution that contains a DC-race, wr(x) \not\prec_{dc} rd(x). The arrows indicate DC ordering between sync(*) events which contain conflicting events. The HB ordered events rel(m)$^{T1}$ $\prec_{hb}$ acq(m)$^{T4}$ compose with the WCP ordered events sync(p)$^{T4}$ $\prec_{wcp}$ sync(p)$^{T5}$ which implies rel(m)$^{T1}$ $\prec_{wcp}$ sync(p)$^{T5}$, thus wr(x) $\prec_{wcp}$ rd(x). Since DC eliminates conservative ordering with the HB relation, wr(x) \not\prec_{dc} rd(x), correctly detecting the predictable race on x shown by the valid reordered trace in Figure 4.1(b).

In fact, DC is complete (as defined in Definition 7): it always detects a DC-race in a trace tr that has a predictable race according to Definition 5.
(a) Example execution. The arrows represent DC ordering (excluding PO ordering).

(b) Correctly reordered trace of (a) demonstrating a predictable race.

Figure 4.1: The execution in (a) has a predictable race and a DC-race ($\text{wr}(x) \not\prec \text{WDC} \text{rd}(x)$) but no WCP-race ($\text{wr}(x) \prec \text{WCP} \text{rd}(x)$). sync(o) is an abbreviation for the sequence $\text{acq}(o); \text{sync}(o); \text{wr}(o\text{Var}); \text{rd}(o\text{Var}); \text{rel}(o)$. 

83
4.3 Completeness

DC is complete: it does not miss predictable races as defined by Definition 5.

Theorem 1 (DC completeness). If a trace tr has a predictable race, then tr has a DC-race.

To prove completeness, we introduce and prove Lemma 1. First, we introduce a concept called DC-distance that defines a finite minimum number of DC rules that must be applied for $e \prec_{DC} e'$:

Definition 12 (DC-distance). The DC-distance of event $e$ and $e'$, $d(e, e')$, is defined as follows (the rules refer to the DC rules in Definition 11):

$$d(e, e') = \min \begin{cases} 0 & \text{if } e \prec_{DC} e' \text{ by rule (a)} \\ 1 + d(A(e), e') & \text{if } e \prec_{DC} e' \text{ by rule (b)} \\ 0 & \text{if } e \prec_{PO} e' \text{ by rule (c)} \\ 1 + \min_{e''} (\max(d(e, e''), d(e'', e'))) & \text{if } \exists e'' \mid e \prec_{DC} e'' \land e'' \prec_{DC} e' \text{ by rule (d)} \\ \infty & \text{otherwise} \end{cases}$$

Note that $d(e, e') = \infty$ if and only if $e \not\prec_{DC} e'$. Like DC, the DC-distance rules feed into each other recursively, e.g., rule (d) satisfies rule (b), which satisfies rule (d), etc. DC-distance converges (provides finite distance for DC-ordered events) by being defined as taking the minimum distance over all choices.

Lemma 1. For any two events $e_1$ and $e_2$ in tr such that $e_1 \prec tr e_2$, if $e_1 \prec_{DC} e_2$ then $e_1$ cannot be reordered after $e_2$.

Proof. Let $e_1$ and $e_2$ be events in tr such that $e_1 \prec tr e_2$. Suppose $e_1 \prec_{DC} e_2$. We prove the lemma by induction on the DC-distance $d(e_1, e_2)$. 

84
**Base case:** \( d(e_1, e_2) = 0 \)

By the definition of DC-distance, \( e_1 \preceq_{dc} e_2 \) by rule (a) or (c) of DC (Definition 11).

**Case 1:** \( e_1 \preceq_{dc} e_2 \) by rule (a).

Then \( e_1 \) is a rel(m) event and \( e_2 \) is a rd(x) or wr(x) event. There must exist \( e \) that is a rd(x) or wr(x) event and \( e' \) that is a rel(m) event, such that \( e \in CS(e_1), e_2 \in CS(e') \), and \( e \simeq e_2 \), according to rule (a) of the DC relation. Figure 4.2(a) depicts this case. Since \( e \simeq e_2 \), by CA of a reordered execution (Definition 4) \( e \) cannot be reordered after \( e_2 \).

Since \( e_2 \in CS(e') \) and no two acquire events of the same lock may be totally ordered without an interleaved release event of the same lock (LS of Definition 4), \( e_1 \) and all events in \( CS(e_1) \) must be reordered after \( e' \) in order for \( e_1 \) to be reordered after \( e_2 \). Since \( e \in CS(e_1) \) and \( e \) cannot be reordered after \( e_2 \), \( e_1 \) cannot be reordered after \( e' \). Therefore, \( e_1 \) cannot be reordered after \( e_2 \).

**Case 2:** \( e_1 \preceq_{dc} e_2 \) by rule (c).

Then \( e_1 \preceq_{po} e_2 \), so \( e_1 \) cannot be reordered after \( e_2 \) by the PO rule of correct reordering (Definition 4).

**Inductive step:** \( d(e_1, e_2) > 0 \)

Suppose the lemma statement holds true for any two events \( e \) and \( e' \) such that \( d(e, e') < d(e_1, e_2) \).

Since \( d(e_1, e_2) > 0 \), by the definition of DC-distance, one or both of the following cases applies.

**Case 1:** \( e_1 \preceq_{dc} e_2 \) by rule (b) and \( d(e_1, e_2) = 1 + d(e, e_2) \) where \( e = A(e_1) \).
Then \( e_1 \) and \( e_2 \) are \( \text{rel}(m) \) events of the same lock. There must exist an \( e \) that is an \( \text{acq}(m) \) event, such that \( e \) is the corresponding acquire to \( e_1 \) (i.e. \( e = A(e_1) \)) and \( e \prec_{\text{dc}} e_2 \), according to rule (b) of the DC relation. Figure 4.2(b) depicts this case.

By the inductive hypothesis, since \( d(e, e_2) < d(e_1, e_2) \), \( e \) cannot be reordered after \( e_2 \). Since no two acquire events of the same lock may be totally ordered without an interleaved release event of the same lock (LS of Definition 4), then \( e_1 \) and all events in \( CS(e_1) \), including \( e \), must be reordered after \( e_2 \) in order for \( e_1 \) to be reordered after \( e_2 \). Since \( e \) cannot be reordered after \( e_2 \), then \( e_1 \) cannot be reordered after \( e_2 \).

**Case 2:** \( e_1 \prec_{\text{dc}} e_2 \) by rule (d) and \( \exists e \mid d(e_1, e_2) = 1 + \max(d(e_1, e), d(e, e_2)) \).

Let \( e \) be such that \( e_1 \prec_{\text{dc}} e \prec_{\text{dc}} e_2 \) and \( d(e_1, e_2) = 1 + \max(d(e_1, e), d(e, e_2)) \). Therefore, \( d(e_1, e) < d(e_1, e_2) \) and \( d(e, e_2) < d(e_1, e_2) \). So by the inductive hypothesis, \( e_1 \)
cannot be reordered after \( e \) and \( e \) cannot be reordered after \( e_2 \). Therefore, \( e_1 \) cannot be reordered after \( e_2 \).

Thus, in all cases \( e_1 \) cannot be reordered after \( e_2 \).

We can now prove that DC is complete.

**Proof of Theorem 1.** Suppose a predictable race (Definition 5) exists in the trace \( tr \) between \( e_1 \) and \( e_2 \) such that \( e_1 < tr e_2 \), but no DC-race exists. Thus \( e_1 \prec_{dc} e_2 \) and therefore, by Lemma 1, \( e_1 \) cannot be reordered after \( e_2 \). Note that \( e_1 \) and \( e_2 \) cannot be directly ordered by rule (a) or (b) of the DC relation (Definition 11), both of which require \( e_1 \), which is a read or write access, to be a release event. Thus one or both of the following hold:

**Case 1:** \( e_1 \) and \( e_2 \) are directly ordered by DC rule (c).

So \( e_1 \prec_{ro} e_2 \) in \( tr \). Thus \( e_1 \) and \( e_2 \) cannot be conflicting accesses and therefore cannot be a predictable race, a contradiction.

**Case 2:** \( e_1 \) and \( e_2 \) are directly ordered by DC rule (d).

So \( \exists e \mid e_1 \prec_{dc} e \prec_{dc} e_2 \) in \( tr \). By Lemma 1, \( e \) cannot be reordered before \( e_1 \) or after \( e_2 \), so \( e_1 < tr' e < tr' e_2 \) for any reordered trace \( tr' \) that includes \( e_2 \). Thus \( e_1 \) and \( e_2 \) cannot be consecutive in any reordered trace, which contradicts the initial assumption of a predictable race between \( e_1 \) and \( e_2 \).

Thus all applicable cases lead to a contradiction.

However, DC is **unsound**: a DC-race may not be a true predictable race. Figures 4.3(a) and 4.3(b) show executions that each have conflicting events on \( x \) that are unordered by DC but cannot be consecutive in any correctly reordered trace.

87
4.4 Doesn’t-Commute Analysis

We introduce DC analysis, a dynamic analysis that tracks the DC relation at every event and detects DC-races. DC analysis (Algorithm 8) computes DC using vector clocks to represent logical time. The analysis is analogous to prior work’s WCP analysis [47]; the differences are related to the fact that DC composes with itself and PO but not HB.

The following notation and terminology follow the WCP paper’s [47]. A vector clock $C: Tid \rightarrow Val$ maps each thread to a nonnegative integer [64]. Operations on vector clocks are point-wise comparison ($\sqsubseteq$) and point-wise join ($\sqcup$):

$$C_1 \sqsubseteq C_2 \iff \forall t.C_1(t) \leq C_2(t)$$

$$C_1 \sqcup C_2 \equiv \lambda t.\max(C_1(t), C_2(t))$$

DC analysis, detailed in Algorithm 8, maintains the following analysis state at each type of event in an execution trace defined in Section 2.1 (page 9):

- a vector clock $C_t$ for each thread $t$ that represents $t$’s current time;
- vector clocks $R_x$ and $W_x$ for each program variable $x$ that represent times of reads and writes, respectively, to $x$;
- vector clocks $L_{m,x}^r$ and $L_{m,x}^w$ that represent the times of critical sections on lock $m$ containing reads and writes, respectively, to $x$;
- sets $R_m$ and $W_m$ of variables read and written, respectively, by each lock $m$’s ongoing critical section (if any); and
- queues of vector clocks $Acq_{m,t}(t')$ and $Rel_{m,t}(t')$, explained below.
Initially, every vector clock maps all threads to 0, except $C_t(t)$ is 1 for all $t$, and every set and queue is empty.

### Algorithm 8

**DC analysis at each event type**

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>procedure ACQUIRE(t, m)</code></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1. <code>foreach t' ≠ t do Acq_{m,t'}(t).Enque(C_t)</code></td>
<td>DC rule (b) (rel–rel ordering)</td>
</tr>
<tr>
<td>3</td>
<td><code>end procedure</code></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td><code>procedure RELEASE(t, m)</code></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1. <code>foreach t' ≠ t do</code></td>
<td>DC rule (b) (rel–rel ordering)</td>
</tr>
<tr>
<td>6</td>
<td>1. while <code>Acq_{m,t'}(t').Front() ⊑ C_t do </code>Acq_{m,t'}(t').Deque()`</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1. <code>C_t ← C_t ∪ Rel_{m,t'}(t').Deque()</code></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td><code>end procedure</code></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td><code>procedure WRITE(t, x)</code></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1. <code>foreach m ∈ HeldLocks(t) do</code></td>
<td>DC rule (a) (conflicting critical sections)</td>
</tr>
<tr>
<td>11</td>
<td>1. <code>C_t ← C_t ∪ (L^r_{m,x} ∪ L^w_{m,x} ∪ C_t)</code></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1. <code>W_m ← W_m ∪ {x}</code></td>
<td>DC rule (a) (conflicting critical sections)</td>
</tr>
<tr>
<td>13</td>
<td>1. <code>check W_x ⊑ C_t</code></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td><code>end procedure</code></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td><code>procedure READ(t, x)</code></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1. <code>foreach m ∈ HeldLocks(t) do</code></td>
<td>DC rule (a) (conflicting critical sections)</td>
</tr>
<tr>
<td>17</td>
<td>1. <code>C_t ← C_t ∪ L^w_{m,x}</code></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>1. <code>R_m ← R_m ∪ {x}</code></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>1. <code>check W_x ⊑ C_t</code></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>1. <code>check R_x ⊑ C_t</code></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>1. <code>W_x(t) ← C_t(t)</code></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td><code>end procedure</code></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td><code>procedure READ(t, x)</code></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>1. <code>foreach m ∈ HeldLocks(t) do</code></td>
<td>DC rule (a) (conflicting critical sections)</td>
</tr>
<tr>
<td>25</td>
<td>1. <code>C_t ← C_t ∪ L^w_{m,x}</code></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>1. <code>R_m ← R_m ∪ {x}</code></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>1. <code>check W_x ⊑ C_t</code></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>1. <code>R_x(t) ← C_t(t)</code></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td><code>end procedure</code></td>
<td></td>
</tr>
</tbody>
</table>

At each release event $r$ executed by $t$, the analysis increments the $t$ logical time $C_t(t)$ (line 13). This action differentiates events that occur before and after $r$, since later the analysis may order $r$ to an event by another thread.

The analysis orders events by DC rule (a) as follows. At a read or write to $x$ by $t$ in a critical section on lock $m$, the analysis joins $C_t$ with all prior critical sections on $m$ that have
performed conflicting events to x (lines 25 and 17). The analysis updates $L_{m,x}^r$ and $L_{m,x}^w$ at each release of $m$ (lines 10–11) using the sets of variables read and written in the critical section on $m$ maintained in $R_m$ and $W_m$ throughout the analysis.

To order events by rule (b) of DC, the analysis uses the queues of vector clocks, $Acq_{m,t}(t')$ and $Rel_{m,t}(t')$, which are arguably the most complex analysis state. $Acq_{m,t}(t')$ is a queue of vector clocks, each of which corresponds to an acquire of lock $m$ by $t'$ that has not been determined to be DC ordered to the most recent release of $m$ by $t$. $Rel_{m,t}(t')$ maintains a queue of vector clocks for the release events corresponding to each acquire event represented in $Acq_{m,t}(t')$.

These queues of vector clocks help to order events by DC rule (b): at $t$’s release of $m$, the analysis checks whether the release is ordered to a prior acquire of $m$ by each thread $t'$, by checking $Acq_{m,t}(t').Front() \subseteq C_t$ (line 6). If so, the algorithm orders $t$’s release of $m$ to the corresponding release of $Acq_{m,t}(t').Front()$ (line 8). It is sufficient to use a queue and check only the head (line 6): other vector clocks in the queue will pass the check only if the head also passes the check (since PO implies DC).

At each read and write, DC analysis maintains the logical time of each thread’s last read and write to x (lines 28 and 21). The analysis checks for DC-races by checking for DC ordering with prior conflicting events to x; a failed check indicates a DC-race (lines 27, 19, and 20).
Figure 4.3: Each execution has a DC-race, i.e., $\text{wr}(x) \not\prec_{\text{DC}} \text{rd}(x)$, but no predictable race. The arrows show DC ordering (excluding PO ordering). $\text{sync}(o)$ is an abbreviation for the sequence $\text{acq}(o); \text{rd}(o\text{Var}); \text{wr}(o\text{Var}); \text{rel}(o)$.

4.5 Weak-Doesn’t-Commute Relation

This section introduces the weak-doesn’t-commute (WDC) analysis that computes the WDC relation, which is weaker than DC and capable of detecting all predictable races but may detect additional false races. WDC analysis is cheaper than DC analysis, and in practice, WDC analysis does not detect many false races, which can be ruled out with vindication (presented in Chapter 5).

Definition 13 (Weak-doesn’t-commute). Given a trace $tr$, $\prec_{\text{WDC}}$ is the smallest relation such that:
(a) If two critical sections on the same lock contain conflicting events, then the first critical section is ordered by WDC to the second conflicting event. That is, \( r_1 \prec_{\text{WDC}} e_2 \) if \( r_1 \) and \( r_2 \) are release events on the same lock, \( r_1 \prec_{\text{tr}} r_2 \), \( e_1 \in \text{CS}(r_1) \), \( e_2 \in \text{CS}(r_2) \), and \( e_1 \simeq e_2 \).

(b) Two events are ordered by WDC if they are ordered by PO. That is, \( e \prec_{\text{WDC}} e' \) if \( e \prec_{\text{PO}} e' \).

(c) WDC is transitively closed. That is, \( e \prec_{\text{WDC}} e' \) if \( \exists e'' \mid e \prec_{\text{WDC}} e'' \land e'' \prec_{\text{WDC}} e' \).

Note that WDC’s rules are identical to DC with the exception of omitting DC rule (b) (Definition 11). An execution trace has a WDC-race if it has two conflicting events that are unordered by the WDC relation. The strict partial order \( \prec_{\text{WDC}} \) is strictly weaker than \( \prec_{\text{DC}} \), and as a result, is high-coverage (as defined in Definition 7) but able to detect more false races than DC.

### 4.6 Weak-Doesn’t-Commute Analysis

We introduce WDC analysis, a dynamic analysis that tracks the WDC relation at every event and detects WDC-races. The analysis is similar to DC analysis, using identical notation and terminology, with the difference being removing lines 2 and 5–9 from Algorithm 8 yields WDC analysis, shown in Algorithm 9.

Although WDC analysis detects more false races than DC analysis, vindication (presented in Chapter 5) can be used without modification to verify if a WDC-ace is a predictable race, making the overall approach sound. Soundness is an important factor when considering the design of a dynamic race detector. The inclusion of WDC analysis contradicts this notion, but Chapter 6 shows the performance benefits and optimization potential of WDC analysis compared to DC analysis.
Algorithm 9  
WDC analysis at each event type

1: procedure ACQUIRE\((t, m)\)  
2: end procedure  

3: procedure RELEASE\((t, m)\)  
4: \[ \text{foreach } x \in R_m \text{ do } L'_{m,x} \leftarrow L'_{m,x} \sqcup C_t \]  
5: \[ \text{foreach } x \in W_m \text{ do } L''_{m,x} \leftarrow L''_{m,x} \sqcup C_t \]  
6: \[ R_m \leftarrow W_m \leftarrow \emptyset \]  
7: \[ C_t(t) \leftarrow C_t(t) + 1 \]  
8: end procedure  

9: procedure WRITE\((t, x)\)  
10: \[ \text{foreach } m \in \text{HeldLocks}(t) \text{ do } \]  
11: \[ C_t \leftarrow C_t \sqcup \{ L'_{m,x} \sqcup L''_{m,x} \} \]  
12: \[ W_m \leftarrow W_m \cup \{ x \} \]  
13: check \( W \subseteq C_t \)  
14: check \( R \subseteq C_t \)  
15: \[ W_x(t) \leftarrow C_t(t) \]  
16: end procedure  

17: procedure READ\((t, x)\)  
18: \[ \text{foreach } m \in \text{HeldLocks}(t) \text{ do } \]  
19: \[ C_t \leftarrow C_t \sqcup L''_{m,x} \]  
20: \[ R_m \leftarrow R_m \cup \{ x \} \]  
21: check \( W \subseteq C_t \)  
22: \[ R_x(t) \leftarrow C_t(t) \]  
23: end procedure  

Examples. The DC-races in Figures 2.2 and 2.4 are also WDC-races. Figure 4.4 shows an execution with no predictable race but with a WDC-race on \( x \). The execution has no DC-race because \( \text{acq}(m)^{T_1} \prec_{\text{dc}} \text{rel}(m)^{T_3} \) implies \( \text{rel}(m)^{T_1} \prec_{\text{dc}} \text{rel}(m)^{T_3} \) by DC rule (b). In contrast, \( \text{rel}(m)^{T_1} \not\prec_{\text{WDC}} \text{rel}(m)^{T_3} \). Thus \( \text{rd}(x)^{T_1} \not\prec_{\text{WDC}} \text{wr}(x)^{T_3} \).
4.7 Contribution and Impact

The DC and WDC relations strengthen race detection capability beyond existing predictive analyses by eliminating the overly conservative composition with the happens-before relation while preserving essential ordering between critical sections. Both DC and WDC analyses, which track the DC and WDC relations respectively, achieve high-coverage partial-order-based predictive analysis by detecting all predictable races from full program executions in time and space that are reasonable for heavyweight in-house testing.

Existing prior work bounds race coverage by the limitation of providing sound predictive analysis. Evaluation, in Chapters 5 and 6, of large Java programs shows that DC and WDC find hard-to-detect predictable races that existing sound predictive analyses cannot find, at a comparable performance cost. Furthermore, at least in our experiments, DC and WDC find all predictable races that are knowable from the observed executions. This dissertation thus addresses an open question of how many predictable races are knowable from real dynamic executions. DC and WDC find several statically distinct predictable races that WCP analysis cannot find, with comparable run-time overhead. By finding all hard-to-detect races in full
program executions, DC and WDC analysis advances the state of the art in predictive race
detection.

Both analyses are unsound with high performance costs compared to highly optimized
HB race detectors. Chapter 5 introduces an approach, Vindicator, that builds a constraint
graph during execution to verify detected DC-races and WDC-races are true races. Chapter 6
considers a set of optimizations, SmartTrack, that improve the performance of partial-order-
based predictive analyses to be competitive with optimized HB analyses.
Chapter 5: Vindicator: Providing Soundness for Unsound High-Coverage Predictive Race Detection

5.1 Introduction

Realizing a practical high-coverage sound predictive analysis requires an analysis that reports only true data races. The DC and WDC analyses, introduced in Chapter 4, detect all predictable races from an observed execution but are unsound analyses that may report false races. Diagnosing a data race—whether true or false—can take weeks to reproduce and fix in production systems [36, 55]. Prior predictive analyses and existing race detectors commonly ensure soundness (report only true races) to provide an effective tool for developers. This chapter presents Vindicator, an approach that extends both DC and WDC analyses to provide the same vital soundness guarantees as existing race detectors. Vindicator presents a novel algorithm that verifies every detected DC-race and WDC-race to report only true data races to achieve high-coverage sound predictive analyses.

Vindicator ensures a sound predictive analysis for each detected data race by building a valid reordered execution trace that exposes conflicting accesses of the data race executing consecutively. Since every verified data race is guaranteed to be a true data race, Vindicator provides soundness without sacrificing high-coverage for DC and WDC analysis. Thus Vindicator provides a high-coverage sound predictive analysis that detects more data races
than existing predictive approaches. The rest of this chapter focuses on DC analysis because (1) the WDC analysis is similar to DC analysis in many respects, and (2) Vindicator can be used directly by the WDC analysis.

5.2 Vindicating Predictable Races

The Vindicator approach presents \texttt{VINDICATE\textsc{race}} in Algorithm 10, an algorithm that determines whether a DC-race or WDC-race is a true predictable race. Note the rest of this chapter focuses on DC analysis since it is straightforward to apply \texttt{VINDICATE\textsc{race}} to WDC. DC analysis identifies \emph{DC-races}, which are \emph{potential} predictable races (i.e., may be false races), and computes a \emph{constraint graph} used by \texttt{VINDICATE\textsc{race}}. \texttt{VINDICATE\textsc{race}} takes as input a single DC-race and a constraint graph, initially consisting of nodes that represent executed events and edges whose reachability correspond to DC order. Intuitively, the constraint graph’s edges serve as constraints on a reordered trace, but the initial constraints (i.e., DC order) are not sufficient to satisfy the rules of a correctly reordered trace that exposes the input race. \texttt{VINDICATE\textsc{race}} thus adds additional needed constraints, both on the order of the DC-race’s events and on critical sections of the same lock.

Ultimately, either the resulting constraint graph has a cycle that implies the DC-race is a false race; \texttt{VINDICATE\textsc{race}} constructs a correctly reordered trace $tr'$ in which the DC-race’s events are consecutive, verifying the DC-race is a true predictable race; or \texttt{VINDICATE\textsc{race}} fails to construct a correctly reordered trace from the constraint graph, which is inconclusive because the construction uses a greedy algorithm.

The rest of this section first defines the constraint graph and then explains \texttt{VINDICATE\textsc{race}} and its helper procedures.
Algorithm 10  
Check if DC-race is a true predictable race

An execution trace is an ordered list of events: \( \langle e, \ldots, e' \rangle \).

The operator \( \oplus \) concatenates two traces:
\[ (e, \ldots, e') \oplus (e'', \ldots, e'''') \equiv (e, \ldots, e', e'', \ldots, e''') \].

1: procedure VINDICATERACE\((G, e_1, e_2)\)
2: \( G \leftarrow \text{ADDCONSTRAINTS}(G, e_1, e_2) \)
3: if \( G = \emptyset \) then
4: return No predictable race
5: else
6: \( tr' \leftarrow \text{CONSTRUCTREORDEREDTRACE}(G, e_1, e_2) \)
7: if \( tr' \neq () \) then \( \triangleright \) Check for non-empty trace
8: return Predictable race witnessed by \( tr' \)
9: else
10: return Don’t know
11: end procedure

12: procedure ADDCONSTRAINTS\((G, e_1, e_2)\)
13: \( C \leftarrow \{ (\text{src}, e_2) \mid (\text{src}, e_1) \in G \} \cup \{ (\text{src}, e_1) \mid (\text{src}, e_2) \in G \} \)
14: \( G \leftarrow G \cup C \)
15: do
16: foreach \((\text{src}, \text{snk}) \in C\) do
17: foreach \((a, r) \mid a \text{ is an acq } \land a \sim_G \text{ src } \land r \text{ is a rel } \land \text{ snk } \sim_G r \land L(a) = L(r)\) do
18: if \((A(r) \sim_G e_1 \lor A(r) \sim_G e_2) \land (a \sim_G e_1 \lor a \sim_G e_2)\) then
19: \( C \leftarrow C \cup \{ (R(a), A(r)) \} \)
20: \( G \leftarrow G \cup \{ (R(a), A(r)) \} \)
21: if \( \exists e \mid (e \sim_G e_1 \lor e \sim_G e_2) \land e \sim_G e\) then
22: return \( \emptyset \) \( \triangleright \) Cycle detected; no predictable race
23: while \( C \) has changed
24: return \( G \)
25: end procedure

26: procedure CONSTRUCTREORDEREDTRACE\((G, e_1, e_2)\)
27: \( R \leftarrow \{ e \mid e \sim_G e_1 \lor e \sim_G e_2 \} \) \( \triangleright \) Reachable events
28: do
29: \( tr' \leftarrow \text{ATTEMPTTOCONSTRUCTTRACE}(G, R, e_1, e_2) \)
30: if \( tr' = (r) \) then \( \triangleright tr' \) contains needed (release) event?
31: \( R \leftarrow R \cup \{ r \} \cup \{ e \mid e \sim_G r \} \)
32: while \( R \) has changed
33: return \( tr' \)
34: end procedure

35: procedure ATTEMPTTOCONSTRUCTTRACE\((G, R, e_1, e_2)\)
36: \( tr' \leftarrow \langle e_1, e_2 \rangle \)
37: while \( R \setminus tr' \neq \emptyset \) do
38: \( next \leftarrow \{ e \in R \setminus tr' \mid (\exists e' \mid (e, e') \in G \land e' \in R \setminus tr') \} \)
39: \( legal \leftarrow \{ e \in \text{next} \mid (e) \oplus tr' \text{ satisfies LS} \} \)
40: if \( legal = \emptyset \) then
41: if \( \exists r \mid (\exists e \in \text{next} \mid e \in CS(r) \land r \notin R \setminus (r) \oplus tr' \text{ satisfies LS})\) then
42: return \( (r) \) \( \triangleright \) Return missing release
43: else
44: let \( e \in \text{legal s.t. } (\exists e' \in \text{legal} \mid e <_{tr} e' \) \( \triangleright \) Select latest legal event in \( tr \) order
45: \( tr' \leftarrow (e) \oplus tr' \) \( \triangleright \) Prepend event to trace
46: return \( tr' \) \( \triangleright \) Return correctly reordered trace
47: end procedure

98
5.2.1 The Constraint Graph

The constraint graph $G$ is a directed graph in which the nodes are the events in $tr$. $G$’s edges, e.g., $(e, e') \in G$, intuitively represent constraints on any reordered trace. We use the notation $e \sim_G e'$ to indicate that $e'$ is reachable from $e$ in $G$:

$$e \sim_G e' \equiv (e, e') \in G \lor \exists e'' | e \sim_G e'' \land e'' \sim_G e'$$

When Vindicator calls \texttt{VINDICATERACE}, $G$ initially has edges that represent DC ordering among events. That is, initially the following property holds:

$$\forall e, e' \in tr \ (e \prec_{dc} e' \iff e \sim_G e')$$

Vindicator constructs the initial $G$ during DC analysis (in addition to tracking the DC relation). Alternatively, DC analysis could record only the execution’s events, and construct $G$ on demand if and when DC analysis detects a DC-race.

Figures 2.4(a), 4.1(a), 4.3(a), and 4.3(b) (pages 18, and 83–91) show initial constraint graphs for executions that have a DC-race but no WCP-race. The arrows in each figure represent edges corresponding to DC rules (a) and (b); the figures do \textit{not} explicitly depict DC rule (b) edges since in these examples all rule (b) edges are already implied by other edges, e.g., Figure 4.3(a)’s rel(m) events are already ordered by a rule (a) edge composed with PO. The figures do \textit{not} explicitly show the PO edges that exist between events by the same thread (DC rule (c)). Graph reachability provides transitivity (DC rule (d)). Note that Figure 2.4(a)’s “WCP” edge is equivalent to the “DC” edge that corresponds to rel(o)$^{T1} \prec_{dc} rd(y)$, where rel(o)$^{T1}$ indicates the rel(o) by Thread 1.

The rest of this section uses those four executions as running examples.
Figure 5.1: After adding consecutive-event constraint. Execution has a predictable race. Figure 2.4(b) (page 18) shows the reordered trace constructed by CONSTRUCTREORDERED-TRACE.

Figure 5.2: After adding consecutive-event constraints (left) and then LS constraints (right). The resulting constraint graph is cyclic, and the execution has no predictable race.
Figure 5.3: After adding consecutive-event and LS constraints. Execution has a predictable race. Figure 4.1(b) (page 83) shows the reordered trace constructed by CONSTRUCTREORDEREDTRACE.

Figure 5.4: After adding consecutive-event and LS constraints. The resulting constraint graph is cyclic, and the execution has no predictable race.
5.2.2 Adding Constraints to $G$

The initial constraint graph $G$ lacks some of the constraints that must exist on a correctly reordered trace. For example, in Figure 2.4(a) (page 18), a reordered trace that executes $\text{wr}(x)$ and $\text{rd}(x)$ consecutively must execute the critical sections on $m$ in a different order from the original trace. Similarly, in Figure 4.1(a) (page 83), a reordered trace that exposes the predictable race must execute the critical sections on $m$ and $n$ in reverse order.

$\text{ADDCONSTRAINTS}$ (called at line 2 in Algorithm 10) adds constraints so that the DC-race's events execute consecutively in the reordered trace, and then discovers and adds constraints on the ordering of critical sections.

Making events consecutive. For $e_1$ and $e_2$ (the input DC-race) to be consecutive in a correctly reordered trace, every event that must execute before $e_1$ or $e_2$ must execute before $e_1$ and $e_2$. Lines 13–14 add consecutive-event constraints to $G$: for each predecessor event $\text{src}$ of $e_1$ (or $e_2$) in $G$, $\text{ADDCONSTRAINTS}$ adds an edge from $\text{src}$ to $e_2$ or $e_1$, respectively.

Figures 5.1–5.4 show the updated constraint graphs for four example executions. Each constraint graph represents consecutive-event constraints as dashed arrows. Note that edges from $e_2$ to $e_1$'s predecessor (assuming $e_1 <_{tr} e_2$) will typically point backward relative to $<_{tr}$ order.

In Figure 5.1, $\text{ADDCONSTRAINTS}$ adds only one consecutive-event constraint edge from $\text{rd}(x)$’s predecessor ($\text{rel}(m)$) to $\text{wr}(x)$; $\text{wr}(x)$ has no predecessors to add edges from. The same situation applies to Figure 5.4. In Figures 5.2 and 5.3, $\text{ADDCONSTRAINTS}$ adds an edge from $\text{wr}(x)$’s lone predecessor to $\text{rd}(x)$, and from $\text{rd}(x)$’s lone predecessor to $\text{wr}(x)$.  

102
ADDCONSTRAINTS adds the new constraint edges not only to $G$ but also to a new set $C$ since these edges will be the starting point for discovering ordering constraints on critical sections.

**Ordering critical sections.** Using the added consecutive-event constraints, ADDCONSTRAINTS identifies and adds ordering constraints on critical sections, called *lock semantics (LS) constraints* (lines 15–23 in Algorithm 10). These constraints have the following form: if two critical sections on the same lock are ordered, at least in part, in $G$, and each critical section is ordered, at least in part, before $e_1$ or $e_2$, then the critical sections must be fully ordered in a correctly reordered trace.

Consider the left half of Figure 5.2, in which there is a path from Thread 2’s $\text{acq}(m)$ to Thread 1’s $\text{rel}(m)$, i.e., $\text{acq}(m)^{T2} \sim_g \text{rel}(m)^{T1}$. Since both acquire events are ordered before at least one of $\text{rd}(x)$ or $\text{wr}(x)$ (in this case, both acquires reach both accesses), we know that at least part of each critical section must execute in $tr'$. ADDCONSTRAINTS identifies such critical sections in lines 17–18. Since these conditions hold, $\text{acq}(m)^{T2}$’s critical section must execute entirely before $\text{rel}(m)^{T1}$’s critical section, to enforce the LS rule of a correctly reordered trace on $tr'$. So ADDCONSTRAINTS adds an edge (dotted arrow) from $\text{rel}(m)^{T2}$ to $\text{acq}(m)^{T1}$ (lines 19–20). There is now a cycle that reaches $\text{wr}(x)$ and $\text{rd}(x)$ (detected at lines 21–22, as discussed below).

In general, the newly added edges may reveal new critical sections that must be fully ordered, and so ADDCONSTRAINTS continues looking for all ordered critical sections from edges in $C$ until convergence.

In Figure 5.3, after ADDCONSTRAINTS adds the consecutive-event edges (dashed arrows), it detects that $\text{acq}(m)^{T4} \sim_g \text{rel}(m)^{T1}$ and that both critical sections reach at least
one access, so it adds an edge from rel(m)^T4 to acq(m)^T1. This edge in turn creates the path acq(n)^T3 \sim_G rel(n)^T2, and adds the edge rel(n)^T3 to acq(n)^T2. After that, ADDCONSTRAINTS finds no new edges to add (convergence), and it returns.

In Figure 5.4, after ADDCONSTRAINTS adds consecutive-event edges, it detects that acq(n)^T3 \sim_G rel(n)^T2 and that both critical sections reach at least one access, so it adds an edge from rel(n)^T3 to acq(n)^T2. After adding this edge, the graph has a cycle that lines 21–22 detect (discussed below).

In Figure 5.1, ADDCONSTRAINTS does not identify any LS constraints to add (thus no dotted arrows in the figure). The consecutive-event edge is sufficient to constrain the reordered trace.

5.2.3 Detecting Cycles in the Constraint Graph

After ADDCONSTRAINTS adds consecutive-event and LS constraints to G, it may be possible to construct a correctly reordered trace that includes e_1 and e_2 and satisfies G’s constraints—but only if G does not have a cycle of constraints ordered before e_1 or e_2. ADDCONSTRAINTS checks this condition (line 21) and returns an empty graph to indicate a cycle (line 22). (A cycle in G that does not reach e_1 or e_2 is not constraining since a correctly reordered trace does not need to contain any events after e_1 and e_2.)

In Figure 5.2, we can see that a cycle exists that reaches e_1 (and e_2): acq(m)^T1 \sim_G rel(m)^T2 \sim_G acq(m)^T1 \sim_G wr(x)^T1. Likewise, in Figure 5.4, a cycle exists that reaches e_2 (and e_1), e.g., acq(n)^T2 \sim_G rel(m)^T4 \sim_G wr(x)^T1 \sim_G rel(n)^T3 \sim_G acq(n)^T2 \sim_G rd(x)^T4. ADDCONSTRAINTS detects these cycles, and VINDICATERACE reports the impossibility of constructing a correctly reordered trace.
In contrast, Figures 5.1 and 5.3 are acyclic, and ADDCONSTRAINTS returns an updated graph \( G \).

**Completeness.** We informally argue, but have not formally proved, that the constraints computed by ADDCONSTRAINTS are *complete*, i.e., ADDCONSTRAINTS detects no cycle if a predictable race exists. Our argument relies on showing that every constraint added by ADDCONSTRAINTS is a necessary constraint on any reordered trace \( tr' \) in which \( e_1 \) and \( e_2 \) execute consecutively:

- Each of ADDCONSTRAINTS’s *consecutive-event constraints* is necessary for executing the conflicting events consecutively on \( tr' \).

- Each of ADDCONSTRAINTS’s *lock semantics (LS) constraints* is necessary, by the following argument. Suppose \( a_1 \) and \( a_2 \) are two acquire events that both must be in \( tr' \), and \( a_2 \leadsto_G R(a_1) \) already exists from a prior step of ADDCONSTRAINTS. Since both \( a_1 \) and \( a_2 \) both must be in \( tr' \), at least one of the critical sections must be in \( tr' \) in its entirety, i.e., either \( R(a_1) <_{tr'} a_2 \) or \( R(a_2) <_{tr'} a_1 \). Inductively assuming that \( a_2 \leadsto_G R(a_1) \) is a necessary constraint on \( tr' \), then \( R(a_1) \not<_{tr'} a_2 \) and therefore \( R(a_2) <_{tr'} a_1 \). Thus adding \((R(a_2), a_1)\) to \( G \) is a necessary constraint on \( tr' \).

Since ADDCONSTRAINTS adds only necessary constraints, and the initial \( G \) has only necessary constraints (since DC is complete), all of \( G \)'s edges are necessary constraints on \( tr' \). Thus a cycle would contradict the existence of \( tr' \) that executes \( e_1 \) and \( e_2 \) consecutively.

**Discussion.** Although cyclic constraint graphs are possible (e.g., Figures 5.2 and 5.4), we have not encountered a cyclic graph in our experiments. Our experiments not only encounter only acyclic constraint graphs, but each graph corresponds to a true predictable race.
However, it is possible for ADDCONSTRANTS to return an acyclic graph even when no predictable race exists; Section 5.3 shows an example execution. Briefly, the example involves two pairs of critical sections on different locks whose implicit dependencies are cyclic.

Regardless, VINDICATERACE is sound overall because, prior to reporting a predictable race, it ensures it can construct a correctly reordered trace, as described next.

### 5.2.4 Constructing a Reordered Trace

Finally, if the computed constraints do not contain a cycle that reaches the input DC-race, VINDICATERACE tries to construct a correctly reordered trace \( tr' \) by calling CONSTRUCT-REORDEREDTRACE (line 6 in Algorithm 10). While \( G \) provides PO and CA ordering and some LS ordering (from Definition 4), it does not totally order all critical sections on the same lock. For example, Figure 5.3 contains neither the path \( \text{rel}(l)^{T2} \leadsto_G \text{acq}(l)^{T5} \) nor the path \( \text{rel}(l)^{T5} \leadsto_G \text{acq}(l)^{T2} \).

Thus an acyclic \( G \) is not sufficient to ensure that a correctly reordered trace exists. Furthermore, CONSTRUCTREORDEREDTRACE is a greedy algorithm that does not backtrack to explore all possible traces, avoiding exponential complexity but risking failure when a correctly reordered trace exists.

**Construction algorithm.** CONSTRUCTREORDEREDTRACE first computes the set of events \( R \) that reach \( e_1 \) or \( e_2 \) (line 27); these events (plus \( e_1 \) and \( e_2 \)) must be in \( tr' \).

CONSTRUCTREORDEREDTRACE then calls ATTEMPTTOCONSTRUCTTRACE (line 29), which builds \( tr' \) in reverse order. It first adds \( e_1 \) and \( e_2 \) to \( tr' \) (line 36). It then selects events from \( R \), one at a time, and prepends them to \( tr' \) until \( tr' \) contains all events from
To prepend an event to $tr'$, $tr'$ prepended with the event must satisfy the constraints in $G$ (line 38) and not violate lock semantics (line 39).

Algorithm 10 omits the detailed logic for checking lock semantics. Briefly, events in a critical section on $m$ cannot be prepended if $m$ is currently held by a different thread, and a critical section on $m$ must be prepended in its entirety if $tr'$ already contains events from another critical section on $m$.

ATTEMPTToCONSTRUCTTRACE is a greedy algorithm that repeatedly chooses one event to prepend to $tr'$ among multiple acceptable events. This choice affects the order of critical sections on the same lock in $tr'$. As line 45 shows, ATTEMPTToCONSTRUCTTRACE always chooses the latest event in $<_{tr}$ order among acceptable events. Our insight here is that the original order of critical sections ($<_{tr}$ order) is most likely to avoid a failure to produce a correctly reordered trace. This insight turns out to be correct in practice: in our experiments, choosing the latest event in $<_{tr}$ order always succeeds, while choosing a different legal event can lead ATTEMPTToCONSTRUCTTRACE to fail. Section 5.3 shows an execution for which choosing the latest event succeeds but choosing another event may fail.

Retrying construction. As mentioned above, if $tr'$ already contains an acq($m$) event, then in order to add an event $e \in CS(r)$ where $r$ is a rel($m$) event, ATTEMPTToCONSTRUCTTRACE must add first add $r$ to $tr'$. However, $r$ may not be in $R$! If ATTEMPTToCONSTRUCTTRACE encounters this case (line 41), it returns the missing event $r$ (line 42). CONSTRUCTREORDEREDTRACE then adds $r$ and events that reach $r$ to $R$ (line 31) and calls ATTEMPTToCONSTRUCTTRACE again. In the worst case, $R$ might be missing release events for each critical section that contains a thread’s last event in $R$, bounding the number of times that CONSTRUCTREORDEREDTRACE can retry ATTEMPTToCONSTRUCTTRACE.
Figure 5.3 helps to illustrate this case. After adding Thread 5’s critical section on \(l\) to \(tr'\), \textsc{AttemptToConstructTrace} cannot legally add Thread 2’s \(\text{sync}(r)\) to \(tr'\) without first adding Thread 2’s \(\text{rel}(l)\)—which is not in \(R\). \textsc{AttemptToConstructTrace} ultimately returns Thread 2’s \(\text{rel}(l)\), and \textsc{ConstructReorderedTrace} adds Thread 2’s \(\text{rel}(l)\) (and \(\text{rel}(n)\)) to \(R\) and again calls \textsc{AttemptToConstructTrace}, which returns the correctly reordered trace shown in Figure 4.1(b).

\textsc{AttemptToConstructTrace} eventually returns either a correctly reordered trace \(tr'\) that demonstrates a predictable race (line 47), or it fails if and when there are no missing release events and no legal events to add to \(tr'\), in which case it returns an empty trace (line 43).

**Discussion.** \textsc{ConstructReorderedTrace} is *sound*: if it returns a reordered trace \(tr'\), it is a correctly reordered trace in which \(e_1\) and \(e_2\) are consecutive. That is, it always fails if no predictable race exists.

\textsc{ConstructReorderedTrace} is *incomplete*: the greedy algorithm can fail even when a predictable race in fact exists. Section 5.3 shows an execution for which \textsc{ConstructReorderedTrace} fails by always choosing the latest event, yet a correctly reordered trace is feasible. \textsc{ConstructReorderedTrace} would be complete if it tried all (exponential in trace length) possible orders satisfying \(G\).

A case we have not yet discussed is that an execution may have a predictable *deadlock* (i.e., if a correctly reordered execution has a deadlock) but not a predictable race. Note that \textsc{WCP} is sound with a deadlock caveat: a \textsc{WCP-race} indicates either a predictable race or a predictable deadlock [47]. In contrast, \textsc{VIndicateRace} inherently reports only predictable
races and will not report predictable deadlocks. Future work might be able to modify VINDICATE RACE to detect sufficient conditions for a predictable deadlock.

5.2.5 Asymptotic Complexity

VINDICATE RACE’s time complexity is polynomial in $N$, the length of $tr$, because every loop’s iteration count is bounded by $G$’s size (nodes plus edges). The polynomial’s degree depends on how VINDICATE RACE is implemented. VINDICATE RACE uses $\Omega(N)$ space for both $G$ and $tr'$.

5.3 Limitations of VINDICATE RACE

This section provides four example execution traces that collectively demonstrate that CONSTRUCT REORDERED TRACE is incomplete and ADD CONSTRAINTS is unsound (both procedures are from Algorithm 10). For all four examples, there is a DC-race, and the constraint graph is acyclic after ADD CONSTRAINTS adds its constraints.

5.3.1 CONSTRUCT REORDERED TRACE

We found that for our evaluated programs, CONSTRUCT REORDERED TRACE always succeeded if it chose the latest legal event, but it sometimes failed if allowed to choose any legal event (Section 5.2.4). We first show examples where choosing an event other than the latest event can fail (i.e., if we change line 45 of CONSTRUCT REORDERED TRACE to “let $e \in legalEvents$” so the algorithm may choose any legal event to prepend at each step), then show an example where choosing the latest event can fail.

Choosing arbitrary events can fail. Figure 5.5 shows an execution for which a correctly reordered execution exists, but CONSTRUCT REORDERED TRACE can fail if it chooses a
Figure 5.5: An execution with a predictable race and a DC-race \((wr(x) \not\prec_{DC} rd(x))\). Solid arrows are the initial DC constraints, and dashed arrows are constraints added by ADDCONSTRAINTS. The circled numbers (e.g., \(\circled{1}\)) represent the order of events prepended to a reordered trace by CONSTRUCTREORDEREDTRACE. A seventh event cannot be prepended to the trace without violating the graph constraints or lock semantics.

Certain legal event at each step. The figure marks six legal events that CONSTRUCTREORDEREDTRACE has so far prepended to \(tr'\). However, there is no seventh legal event to prepend to \(tr'\). Since \(acq(m)^T1 <_{tr'} rel(p)^T2\) and \(acq(p)^T3 <_{tr'} rel(m)^T4\), the remaining mandatory constraints \(rel(m)^T4 <_{tr'} acq(m)^T1\) and \(rel(p)^T2 <_{tr'} acq(p)^T3\) are impossible to satisfy since \(rel(m)^T4 <_{tr'} acq(m)^T1 <_{tr'} rel(p)^T2 <_{tr'} acq(p)^T3 <_{tr'} rel(m)^T4\). We see that CONSTRUCTREORDEREDTRACE is a greedy algorithm that chooses events without considering remaining implicit constraints, resulting in failure in this case.

Similarly, Figure 5.6(a) (i.e., Figure 5.6 excluding the sync(s6) events) shows an execution for which CONSTRUCTREORDEREDTRACE can fail if it does not choose the latest event. In particular, choosing \(rel(p)^T3\), or both \(rel(p)^T5\) and \(rel(m)^T2\), prematurely (before a later legal event) will fail to construct a correctly reordered trace.
Figure 5.6: We use this figure to represent three different example executions: (a) an execution omitting the sync(s6) events, which has a predictable race; (b) an execution omitting the sync(s4) events, which has a predictable race; and (c) the unmodified execution, which has no predictable race. The arrows represent DC ordering.

We note that for both executions, if \textsc{construct\textsc{reordered\textsc{trace}}} always chooses the latest event, it constructs a reordered trace successfully.

**Choosing the latest event can fail.** Consider Figure 5.6(b) (i.e., Figure 5.6 excluding the sync(s4) events). (Unmodified) \textsc{construct\textsc{reordered\textsc{trace}}} chooses the latest legal event at each step, which ultimately leads to failure. In particular, prepending rel(p)^{T5} and rel(m)^{T4} before the other critical sections on p and m makes failure inevitable. Although
choosing the latest legal event fails, there does exist a set of legal event choices that
\texttt{CONSTRUCTREORDEREDTRACE} can make to construct a correctly reordered trace.

\subsection{ADD\texttt{CONSTR}RAINTS}

The execution in Figure 5.6(c) (i.e., the figure exactly as shown) has no predictable race. However, \texttt{ADD\texttt{CONSTR}AIN\texttt{T}} produces an acyclic graph for this trace. Intuitively, there exist implicit, mutually incompatible ordering constraints on two pairs of critical sections on different locks, for which \texttt{ADD\texttt{CONSTR}AIN\texttt{T}} does not produce a cycle of dependencies.

Although \texttt{ADD\texttt{CONSTR}AIN\texttt{T}} is unsound, \texttt{CONSTRUCTREORDEREDTRACE} is sound and thus \texttt{VINDICATE\texttt{R}\texttt{ACE}} is sound. \texttt{CONSTRUCTREORDEREDTRACE} fails to construct a correctly reordered trace for Figure 5.6(c), and \texttt{VINDICATE\texttt{R}\texttt{ACE}} returns “Don’t know” (Algorithm 10).

\section{Evaluation}

This section evaluates Vindicator’s ability to detect predictable races and its run-time performance, compared with competing approaches. Our evaluation focuses on Vindicator applied to DC analysis since it is straightforward to apply Vindicator to WDC and WDC analysis only detects more, if any, false races than DC analysis.

\subsection{Implementation}

We implemented Vindicator in \textit{RoadRunner} version 0.5 as discussed in Section 2.4 (page 20). Our implementation of Vindicator is publicly available.\textsuperscript{10}

Our Vindicator implementation has two main components: DC analysis and \texttt{VINDICATE\texttt{R}\texttt{ACE}}. DC analysis constructs the constraint graph \(G\) as it executes, storing it in memory.

\textsuperscript{10}https://github.com/PLaSSticity/Vindicator.git
and logs all detected DC-races. Vindicator performs WCP and HB analyses alongside DC analysis to determine if each DC-race is also a WCP-race and/or an HB-race.

When the program execution ends, Vindicator calls \texttt{VINDICATERACE} on each \textit{DC-only race}, which is a DC-race that is not also a WCP-race. Since every WCP-race is a DC-race and every WCP-race is a true predictable race or deadlock [47], only DC-races that are not WCP-races need to be vindicated to verify true predictable races. The implementation supports optionally calling \texttt{VINDICATERACE} on a WCP-race to obtain a correctly reordered trace or to distinguish a race from a deadlock.

\textbf{DC analysis and constraint graph construction.} Our implementation of DC analysis handles read, write, acquire, and release events according to Algorithm 8 in Section 4.4 (Chapter 4). As Section 2.4 describes, Vindicator supports additional synchronization primitives by joining vector clocks and updating $G$ accordingly. Two evaluated programs (pmd and tomcat) fork threads implicitly, using constructs from \texttt{java.util.concurrent} instead of explicitly invoking \texttt{Thread.start()}. Since RoadRunner does not currently support interposing on these constructs, DC analysis detects thread start and terminate in these cases and conservatively adds thread fork and join edges.

For each event $e$ that DC analysis handles, the analysis creates an event node $e$ in the constraint graph $G$, and adds edge(s) to $G$ for newly established DC ordering(s). DC analysis minimizes the number of edges added to $G$, adding $(e_{src}, e)$ to $G$ only if $e_{src} \sim_G e$ does not already hold, using DC analysis vector clocks to determine which ordering(s) have been newly established at $e$. To determine each $e_{src}$, DC analysis tracks, for each variable $x$ and synchronization object $m$, the last event by each thread that accessed $x$ or $m$. 

113
DC analysis processes parallel events in parallel, using fine-grained synchronization on analysis metadata and nodes in $G$ to provide analysis atomicity without serializing the analysis. Since the analysis does not observe a total order $<_{tr}$ of events, it assigns each event node a Lamport timestamp [49] such that $e \prec_{HB} e' \implies ts(e) < ts(e')$.

DC analysis uses an instrumentation “fast path” that identifies and skips redundant accesses (if there is a prior write, or if the prior and current events are reads) to the same variable by the same thread without interleaving synchronization. The fast path reduces run-time overhead and the size of $G$. Reducing $G$’s size not only reduces space overhead, but it improves VINDICATE RACE’s run time. To further reduce the size of $G$, DC analysis merges adjacent PO-ordered nodes on the fly, i.e., $(e, e') \in G$ becomes a single node, if $e$ and $e'$ are both read and write events and $e$ has no other outgoing edges and $e'$ has no other incoming edges.

**WCP and HB analyses.** In addition to performing DC, WCP, and HB analyses on the same observed trace. We implemented WCP and HB analyses in RoadRunner (instead of using the WCP authors’ available implementation [47] or the FastTrack implementation in RoadRunner [31, 35]) to provide a fair comparison with DC analysis and to identify which DC-races are DC-only races. Each analysis’s time complexity is linear in the size of the trace.

**Handling DC-races.** When DC analysis detects a DC-race between two events $e_1$ and $e_2$, as described in Section 2.4, it updates vector clocks and $G$ so that $e_1 \prec_{dc} e_2$ and $e_1 \sim_G e_2$. Note that every HB-race is a WCP-race, and every WCP-race is a DC-race.

At a read or write event $e$, as described in Section 2.4, DC analysis may detect multiple DC-races with prior read or write events $e'$ such that $e' \not\sim_{dc} e$. DC analysis records only the
“shortest” DC-race, i.e., \( e' \not\prec_{\text{dc}} e \) such that \( e' \) is maximal (potentially choosing arbitrarily among multiple concurrent events).

**VINDICATERACE.** Vindicator calls **VINDICATERACE** (Algorithm 10) on each DC-only race separately, using graph traversals to compute reachability between events. Before **VINDICATERACE** returns, it removes all edges that it added to \( G \), in order to check each DC-race independently.

As a sanity check that is not required for Vindicator’s correctness, the implementation optionally checks that \( tr' \) returned by **CONSTRUCTREORDEREDTRACE** is a correctly reordered trace according to Definition 4.

**VINDICATERACE** uses two correctness-preserving optimizations. First, **ADDCONSTRAINTS** exploits PO ordering among events by the same thread to avoid considering many redundant acquire–release pairs. Second, **ADDCONSTRAINTS** only considers events within a window of events between \( e_1 \) and \( e_2 \), based on events’ Lamport timestamps. To preserve correctness, **ADDCONSTRAINTS** expands the window on the fly to include each edge it adds to \( G \).

### 5.4.2 Methodology

We evaluate Vindicator using the DaCapo benchmarks as discussed in Section 2.5 (page 22).

### 5.4.3 Detection Coverage of Predictable Races

Here we evaluate Vindicator’s predictable race coverage compared with HB and WCP analyses.
Table 5.1: HB-, WCP-, and DC-races detected by our implementation. *DC-races* include all WCP-races; *WCP-races* include all HB-races. **VINDICATE RACE** confirmed that every **DC-race is a true predictable race.** Each result is the average of 10 trials, rounded to the nearest integer.

<table>
<thead>
<tr>
<th>Program</th>
<th>HB-races</th>
<th>WCP-races</th>
<th>DC-races</th>
</tr>
</thead>
<tbody>
<tr>
<td>avrorra</td>
<td>5 (933)</td>
<td>5 (934)</td>
<td>5 (996)</td>
</tr>
<tr>
<td>batik</td>
<td>0 (0)</td>
<td>0 (0)</td>
<td>0 (0)</td>
</tr>
<tr>
<td>h2</td>
<td>10 (990)</td>
<td>11 (793)</td>
<td>11 (1,027)</td>
</tr>
<tr>
<td>jython</td>
<td>3 (3)</td>
<td>3 (4)</td>
<td>3 (4)</td>
</tr>
<tr>
<td>luindex</td>
<td>1 (1)</td>
<td>1 (1)</td>
<td>1 (1)</td>
</tr>
<tr>
<td>lusearch</td>
<td>0 (0)</td>
<td>0 (0)</td>
<td>0 (0)</td>
</tr>
<tr>
<td>pmd</td>
<td>4 (13)</td>
<td>4 (13)</td>
<td>5 (23)</td>
</tr>
<tr>
<td>sunflow</td>
<td>2 (8)</td>
<td>2 (10)</td>
<td>2 (14)</td>
</tr>
<tr>
<td>tomcat</td>
<td>109 (4,604)</td>
<td>110 (4,659)</td>
<td>110 (4,677)</td>
</tr>
<tr>
<td>xalan</td>
<td>4 (16)</td>
<td>63 (3,420)</td>
<td>67 (4,660)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>138 (6,268)</strong></td>
<td><strong>199 (9,834)</strong></td>
<td><strong>204 (11,402)</strong></td>
</tr>
</tbody>
</table>

Table 5.1 reports races detected by DC, WCP, and HB analyses on the same trace. For each kind of race, the table reports statically distinct races (a statically distinct race is an unordered pair of static program source locations), followed by dynamic races (a dynamic race is a pair of events in the trace; multiple dynamic races may correspond to the same statically distinct race) in parentheses, averaged over 10 trials.

The table shows that on average DC analysis reports five static DC-only races, i.e., statically distinct DC-races that are not WCP-races. In addition, there are four static DC-only races (in h2, pmd, and xalan) that each occurs as a static DC-only race in 1 or 2 out of 10 trials, so they are not shown in the table due to rounding.

Table 5.2 shows details of the nine static DC-only races detected across the 10 trials. These static DC-only races did *not* manifest as a WCP-race in any of the trials, except for both of h2’s races and xalan’s `AttributeIterator.getNextNode():56–LocPathIterator.setRoot():372` race, which manifest as a WCP-race in at least one trial. The table shows each statically distinct race’s two static source locations and the range of event distances across all dynamic...
<table>
<thead>
<tr>
<th>Program</th>
<th>Static DC-only race</th>
<th>Event distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>h2</td>
<td>StringCache.getNew():93&lt;br&gt;StringCache.get():48</td>
<td>11,288–248,799</td>
</tr>
<tr>
<td>h2</td>
<td>StringCache.getNew():83&lt;br&gt;StringCache.get():54</td>
<td>12,438–14,182</td>
</tr>
<tr>
<td>pmd</td>
<td>PMD.getSourceTypeOfFile():152&lt;br&gt;PMD.&lt;init&gt;():57</td>
<td></td>
</tr>
<tr>
<td>pmd</td>
<td>PMD.setExcludeMarker():234&lt;br&gt;PMD.processFile():96</td>
<td></td>
</tr>
<tr>
<td>xalan</td>
<td>AttributelIterator.getNextNode():56&lt;br&gt;LocPathIterator.setRoot():372</td>
<td></td>
</tr>
<tr>
<td>xalan</td>
<td>FastStringBuffer.&lt;init&gt;():210&lt;br&gt;FastStringBuffer.append():488</td>
<td>2,146–2,629,775</td>
</tr>
<tr>
<td>xalan</td>
<td>OneStepIterator.setRoot():97&lt;br&gt;OneStepIterator.setRoot():97</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2: Characteristics of the nine static DC-only races reported by Vindicator in our experiments. Each race is an unordered pair of static locations, which are represented as class, method, and line number. Event distance is the range of event distances (distance apart in $<_{tr}$ between the two conflicting events) across all dynamic instances of the race, from a separate experiment that totally orders events.

instances of the DC-only race across all trials in which it occurred. A dynamic race’s event distance is the number of events apart, in the observed trace order $<_{tr}$, that the two conflicting events occurred. Since the implementation does not totally order events (Section 5.4.1), we collected event distances in a separate 10 trials that use global synchronization to assign totally ordered timestamps. In these 10 separate trials, five of Table 5.2’s nine races occurred as static DC-only races.

Several programs have dynamic DC-only races, i.e., dynamic DC-races that are not also WCP-races. For example, the table reports 62 dynamic DC-only races on average for avrora. However, each of these races maps to the same statically distinct race as some dynamic WCP-race in the same trial, so avrora has no static DC-only races.
Figure 5.7: Cumulative distribution of the event distance of three kinds of dynamic races. For a given event distance, the plot shows the percentage of dynamic races with at least that event distance.

**Event distances of all dynamic races.** Figure 5.7 plots the cumulative distribution of event distances of all dynamic races from the 10 separate, globally synchronized trials described above. Each dynamic race is either a DC-only race, a WCP-only race (a WCP-race that is not an HB-race), or an HB-race, and appears exactly once in the plot. For any event distance, the plot shows the percentage of dynamic races that have at least that event distance.

The plot shows that DC-only races have larger event distances than HB-races or WCP-only races by an order of magnitude or more. This result is notable for two reasons. First,
prior work that is complete cannot scale beyond bounded windows of execution (Chapter 7) and would thus have difficulty finding many DC-only races. Second, VINDICATERACE successfully analyzes every dynamic DC-only race (Section 5.4.4) despite their large event distances.

**Vindicating DC-races.** By default, the Vindicator implementation invokes VINDICATERACE on each dynamic DC-only race, i.e., the difference between dynamic DC- and WCP-races in Table 5.1. In our experiments, VINDICATERACE confirms that every dynamic DC-only race is a true predictable race. That is, for every dynamic DC-race in Table 5.1 that is not also a WCP-race, VINDICATERACE verifies that it is a true predictable race: VINDICATERACE never encounters a cycle nor fails to construct a reordered trace, and it always reaches line 8 in Algorithm 10, returning a correctly reordered trace $tr'$ that exposes the predictable race. (As a sanity check, our experiments also run VINDICATERACE on every WCP-only and HB-race, always producing a correctly reordered trace that exposes a race.)

Table 5.3 shows characteristics of VINDICATERACE analyzing each of the dynamic DC-only races from the 10 trials. The table reports the distribution across all dynamic DC-only races, i.e., across all calls to VINDICATERACE. LS constraints added is the number of lock semantics (LS) constraints (edges) that ADDCONSTRAINTS adds (lines 19–20 in Algorithm 10). Outer loop iterations is the number of executed iterations of the do–while loop in ADDCONSTRAINTS (lines 15–23).

The table shows that for most dynamic DC-only races, ADDCONSTRAINTS adds no LS constraints and consequently does not repeat its outer loop. (ADDCONSTRAINTS always
Table 5.3: Characteristics of VINDICATE-RACE (Algorithm 10) for all dynamic DC-only races across the 10 trials.

<table>
<thead>
<tr>
<th>LS constraints added</th>
<th>0</th>
<th>1</th>
<th>2–3</th>
<th>4–7</th>
<th>8–15</th>
<th>16–135</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC-only races</td>
<td>14,398</td>
<td>553</td>
<td>325</td>
<td>212</td>
<td>149</td>
<td>32</td>
</tr>
<tr>
<td>Outer loop iterations</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6–14</td>
</tr>
<tr>
<td>DC-only races</td>
<td>14,398</td>
<td>942</td>
<td>158</td>
<td>132</td>
<td>26</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 5.3: Characteristics of VINDICATE-RACE (Algorithm 10) for all dynamic DC-only races across the 10 trials.

adds consecutive-event constraints; lines 13–14.) But for more than 1,000 DC-only races, ADDCONSTRAINTS performs multiple loop iterations and adds several LS constraints.

5.4.4 Run-Time Performance

Table 5.4 shows execution time, memory usage, and other dynamic characteristics of Vindicator, compared with configurations that perform a subset of Vindicator’s functionality. Each value is the arithmetic mean from the 10 trials used in the rest of the evaluation. Events is total executed program events (memory accesses and synchronization operations); the subset of events that are not filtered out by fast-path instrumentation is in parentheses. #Thr is the total number of threads created and, in parentheses, the maximum number of threads active at any time. Reported execution times are wall-clock times. Memory usage is the maximum memory usage across all full-heap garbage collections (GCs) in the execution. Some executions perform no full-heap GCs because the JVM sets the initial heap size to 2 GB, so the table reports only values for full-heap GCs that exceed 2 GB. The Base and Empty configurations never exceed 2 GB, so we omit them from the table.

Base time reports execution time of an uninstrumented program. Other execution times are normalized to Base time. Empty executes RoadRunner’s Empty tool, which instruments programs to generates events, but performs no analysis on them. The WCP configuration performs only WCP analysis, which includes HB analysis. DC performs DC analysis in
<table>
<thead>
<tr>
<th>Program</th>
<th>Events</th>
<th>#Thr</th>
<th>Base time</th>
</tr>
</thead>
<tbody>
<tr>
<td>avrora</td>
<td>1,400M (160M)</td>
<td>7 (7)</td>
<td>2.1 s</td>
</tr>
<tr>
<td>batik</td>
<td>160M (12M)</td>
<td>8 (7)</td>
<td>2.6 s</td>
</tr>
<tr>
<td>h2</td>
<td>3,800M (460M)</td>
<td>10 (9)</td>
<td>4.7 s</td>
</tr>
<tr>
<td>jython</td>
<td>230M (82M)</td>
<td>2 (2)</td>
<td>2.2 s</td>
</tr>
<tr>
<td>luindex</td>
<td>400M (45M)</td>
<td>3 (3)</td>
<td>1.1 s</td>
</tr>
<tr>
<td>lusearch</td>
<td>1,400M (190M)</td>
<td>10 (10)</td>
<td>1.1 s</td>
</tr>
<tr>
<td>pmd</td>
<td>200M (25M)</td>
<td>9 (9)</td>
<td>1.2 s</td>
</tr>
<tr>
<td>sunflow</td>
<td>9,700M (760M)</td>
<td>17 (9)</td>
<td>1.7 s</td>
</tr>
<tr>
<td>tomcat</td>
<td>44M (18M)</td>
<td>35 (22)</td>
<td>0.88 s</td>
</tr>
<tr>
<td>xalan</td>
<td>610M (260M)</td>
<td>9 (9)</td>
<td>2.1 s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Slowdown (normalized to Base time)</th>
<th>Memory usage (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vindicator</td>
<td>WCP DC Vindicator</td>
</tr>
<tr>
<td>avrora</td>
<td>3.2×</td>
</tr>
<tr>
<td>batik</td>
<td>3.7×</td>
</tr>
<tr>
<td>h2</td>
<td>6.3×</td>
</tr>
<tr>
<td>jython</td>
<td>5.4×</td>
</tr>
<tr>
<td>luindex</td>
<td>5.1×</td>
</tr>
<tr>
<td>lusearch</td>
<td>8.7×</td>
</tr>
<tr>
<td>pmd</td>
<td>5.1×</td>
</tr>
<tr>
<td>sunflow</td>
<td>10×</td>
</tr>
<tr>
<td>tomcat</td>
<td>3.9×</td>
</tr>
<tr>
<td>xalan</td>
<td>2.9×</td>
</tr>
</tbody>
</table>

Table 5.4: Run time and memory usage for Vindicator and other analysis configurations. All values (except thread counts) are rounded to 2 significant figures. The table reports Vindicator’s slowdown relative to uninstrumented execution as the first value in the Vindicator column (e.g., 33× for avrora). The text explains other values.

In addition to WCP analysis, but does not construct the constraint graph G. The table shows that WCP adds substantial run-time and memory overhead. On top of WCP, DC adds moderate run-time overhead but sometimes adds high memory overhead. The memory overhead is expected as even though DC is similar algorithmically to WCP, each analysis maintains separate data structures.

The Vindicator configuration computes all relations while also building the constraint graph; when the program terminates, Vindicator calls VINDICATERACE on each DC-race. Each cell reports three numbers: Vindicator’s run time normalized to Base time, which includes calling VINDICATERACE on a single dynamic instance of each static DC-only
race (shown in parentheses as non-normalized run time), followed by the additional non-normalized run time for checking all remaining dynamic DC-only races (i.e., every dynamic DC-only race not already checked as a static DC-only race). For example, Vindicator takes $91 \times$ longer than 2.1 s to analyze xalan, including 11 s to run VINDICATE RACE, on average, on 4 static DC-only races. It takes an additional 2,400 s to run VINDICATE RACE on 1,236 additional dynamic DC-races. See Table 5.1 for corresponding static and dynamic race counts.

Building the constraint graph $G$ adds relatively low run-time overhead over DC, and often adds low memory overhead because merging event nodes (Section 5.4.1) reduces $G$'s size significantly. Vindicator adds high memory overhead over DC for xalan, which we have confirmed is due to building the constraint graph, not from running VINDICATE RACE.

In a few cases, building $G$ results in lower memory overhead than not building $G$. This counterintuitive result is due to the way we measure memory overhead: by recording the maximum of reported live memory across all full-heap GCs, which is an imperfect estimate of maximum live memory size. As a result, the reported memory is affected by how often GC happens and by the JVM’s automatic adjustments to the heap size between GCs. RoadRunner periodically cleans up analysis resources that are no longer in use, further affecting the measurements.

The table shows that a small fraction of Vindicator’s run time is from calling VINDICATE RACE on static DC-only races (the times in parentheses). Vindicator incurs this overhead only for the three programs that have static DC-only races. Vindicator takes 11 seconds on average to analyze 4 static DC-only races for xalan, and a fraction of a second for 1 static DC-only race for pmd. For h2, only 3 out of 10 trials have a static DC-only race, which VINDICATE RACE takes 14 seconds on average to analyze (or 4.3 seconds on average
across all 10 trials, as the table reports). VINDICATERACE’s relatively high cost for h2 is mainly due to CONSTRUCTREORDEREDTRACE: the races’ event distances are not large, but the events occur late in the execution, so the size of the reordered trace is large, and in fact most of the execution time is GC time. The additional cost of analyzing every dynamic DC-only race (the time values right of the ‘+’ symbol) is nontrivial—a few programs have many dynamic DC-only race—but the average cost of vindicating each race is low.

These results show that, in practice, VINDICATERACE is efficient enough for testing-time use, including for finding and vindicating DC-races whose accesses are millions of events apart.

5.5 Contribution and Impact

Vindicator advances the state of the art in sound predictive race detection by detecting all predictable races from full program executions in time and space comparable to existing sound predictive analyses. Vindicator detects and verifies hard-to-detect races between accesses that are millions of events apart—outside the range of windowed approaches—and also detects and verifies races that the state-of-the-art unbounded approach (WCP) cannot find.

Regarding coverage, Vindicator detects more predictable races than existing approaches. It finds more races than WCP analysis [47], which has the highest coverage among existing sound predictive analyses that scale to full execution traces. Existing approaches that predict more races than WCP rely on constraint solving and cannot scale beyond bounded windows of execution (e.g., [40, 80, 85] (Chapter 7). Our results show that many DC-only races’ accesses are millions of events apart—well outside the range of windowed approaches.
Furthermore, at least in our experiments, Vindicator detects *all* predictable races (according to Definition 5) in real program executions. This work thus helps answer an open question of just how many predictable races exist in real programs.

Our implementation of Vindicator adds significant performance costs, but the overheads are likely acceptable for heavyweight testing and worth the cost to expose new, hard-to-detect data races. The implementation of VINDICATE-RACE is efficient enough to produce reordered traces that verify races between accesses separated by millions of events.

However, Vindicator still takes time that is hundreds of times greater than the native execution of a program. The next chapter aims to close the performance gap between partial-order-based predictive analysis and optimized HB analysis to provide a high-coverage practical predictive analysis.
Chapter 6: SmartTrack: Optimizing Predictive Race Detection

6.1 Introduction

This dissertation aims to develop a practical high-coverage predictive analysis that improves the run time and memory usage performance of predictive analyses to close the performance gap between predictive analysis and highly optimized HB analysis. The partial-order-based predictive analyses WCP (introduced in Chapter 2), DC, and WDC (both introduced in Chapter 4) detect more races than HB analysis. However, these analyses remain impractical, slowing programs by tens or hundreds of times compared with the native execution of the program. The performance acceptable for heavyweight in-house testing becomes a crucial limitation of predictive race detectors when analyzing large real-world programs that execute for hours natively. The contributions of this chapter improve the practicality of predictive analyses and achieve run time and memory usage competitive with highly optimized HB analysis.

Predictive analyses maintain access metadata similar to HB analysis but also maintain lock metadata to track conflicting critical sections. Partial orders weakening the observed order of execution events more than HB analysis must guarantee correct reasoning about the reordering of the observed behaviors. For partial orders to maintain this property, critical sections containing conflicting shared-memory accesses must be ordered, thus requiring
tracking new lock metadata. The additional lock metadata imposes significant time and space complexity for predictive analyses compared to dynamic analysis. The WCP, DC, and WDC analyses are substantially slower than optimized HB analysis (roughly 25–32 × vs. 8 × in prior work [31, 35, 47, 79] and Section 6.4) in part due to the use of full vector clocks to maintain access metadata. Though detecting conflicting critical sections, the vital component of partial-order-based predictive analysis, adds an equivalent cost to maintain additional lock metadata.

This chapter presents SmartTrack, an algorithm applicable to several predictive analyses. SmartTrack’s optimizations enable predictive analyses to perform competitively with highly optimized HB analyses. SmartTrack consists of two synergistic optimizations: epoch and ownership optimizations, which are adapted from prior work optimizing HB analysis [31, 35, 100], and novel conflicting critical section optimizations, which specifically target predictive analysis.

Evaluation of large Java programs shows that SmartTrack’s optimizations improve the performance of predictive analyses substantially compared with prior work. Furthermore, the resulting predictive analyses perform competitively with the state-of-the-art FastTrack analysis [31, 35]. This work shows that predictive analyses can provide practical, high-coverage data race detection by achieving performance on par with the latest highly optimized HB race detectors.

### 6.2 Limitations of Existing Race Detection Analyses

This chapter aims to improve run time performance and memory overhead of existing predictive analyses. Note, this chapter uses the same execution model introduced in Chapter 2 (page 19) and discusses the limitations of existing race detection analyses.
Classic HB analysis computes the HB relation over an execution program and detects HB-races using vector clocks [64]. FastTrack is an optimized state-of-the-art HB analysis [31], and recent work provides additional improvements to FastTrack [35, 100], which are covered later in the chapter. FastTrack’s epoch optimization recognizes that a more lightweight read and write metadata representation is sufficient, resulting in a $2.3 \times$ speedup over high-performance full vector clock based analysis [31]. Discussed further in section 6.3, SmartTrack applies this optimization directly to partial-order-based predictive analyses and achieve similar speedup over unoptimized predictive analysis. However, a significant performance gap remains in tracking the information for conflicting critical sections.

HB analysis has performance acceptable for regular in-house testing—roughly $8 \times$ slowdown according to prior work [31, 35] and our evaluation—and is widely used in data race detector including Google’s ThreadSanitizer [86, 87] and Intel Inspector [43].

While HB analysis has decent performance, it misses data races that are knowable from an analyzed execution trace. As discussed extensively in Chapter 5, Weak-Causally-Precedes (WCP), Doesn’t-Commute (DC), and Weak-Doesn’t-Commute (WDC) have corresponding predictive analyses capable of detecting more races from a single execution than HB analysis but incur significantly higher performance costs than optimized HB analysis.

To better understand the performance costs, we review the details of the DC analysis in Algorithm 8 (introduced in Chapter 4). This chapter focuses on DC analysis because (1) WCP and WDC analyses are similar to DC analysis in many respects, and we find that WCP and WDC analyses are slow for the same reasons as DC analysis, and (2) our optimizations to DC analysis apply directly to both WCP and WDC analysis.
DC analysis performance costs

This section explores the performance costs of the unoptimized DC analysis (Algorithm 8, page 89) introduced in Chapter 4.

A significant and challenging source of performance costs is the logic for detecting conflicting critical sections to provide DC rule (a), a cost not present in HB analysis. At each rel(m), the algorithm updates $L_{\text{m,x}}^r$ and $L_{\text{m,x}}^w$ based on the variables accessed in the latest critical section on m (lines 10–11). At a read or write to x by t, the algorithm uses $L_{\text{m,x}}^r$ and $L_{\text{m,x}}^w$ to join $C_t$ with all prior critical sections on m that performed conflicting accesses to x (lines 17 and 25).

At each write or read to x, the algorithm applies rule (a) as described above, checks for races, and updates the logical time of the last write or read to x. The algorithm checks for DC-races by checking for DC ordering with prior conflicting accesses to x; a failed check indicates a DC-race (lines 19, 20, and 27). The algorithm finally updates the logical time of the current thread’s last write or read to x (lines 21 and 28).

To order events by DC rule (b), the algorithm uses $Acq_{\text{m,t}}(t')$ and $Rel_{\text{m,t}}(t')$ to detect acquire–release ordering between two critical sections and add release–release ordering. Each vector clock in the queue $Acq_{\text{m,t}}(t')$ represents the time of an acq(m) by t’ that has not been determined to be DC ordered to the most recent release of m by t. Vector clocks in $Rel_{\text{m,t}}(t')$ represent the corresponding rel(m) times for clocks in $Acq_{\text{m,t}}(t')$. At rel(m) by t, the algorithm checks whether the release is ordered to a prior acquire of m by any thread t’ (line 6). If so, the algorithm orders the release corresponding to the prior acquire to the current rel(m) (line 8).

Since DC is unsound, a DC-race may not be a predictable race, although DC-races are generally predictable races in practice. Pairing DC analysis with the vindication algorithm...
introduced in Chapter 5 enables checking whether a DC-race is a predictable race. (WCP is sound, so it does not need or use vindication [47].)

Performance Costs of Existing Predictive Analyses

The unoptimized algorithm for DC (and WCP and WDC) analyses incurs two main costs over the FastTrack algorithm for HB analysis [31, 35, 100]: using vector clocks to represent DC times and computing DC rule (a). The third main cost for DC (and WCP) is computing DC rule (b) which the WDC analysis avoids entirely.

DC rule (b). Computing DC rule (b) requires complex queue operations (lines 2 and 5–9 in Algorithm 8),\(^{11}\) at every synchronization operation. As our evaluation of the WDC analysis shows, the analysis to provide DC rule (b) adds run-time overhead but is not DC analysis’s primary run-time cost because it occurs only at lock events, which are infrequent compared with read and write events.

Vector clocks. The updates to write and read metadata and race checks (lines 19–21 and 27–28 in Algorithm 8) are analogous to work performed by HB analysis. However, unlike FastTrack’s optimized HB analysis, which uses epoch optimizations to achieve constant time and space in many cases [31], unoptimized DC analysis uses full vector clock operations for race checks, which take \(O(T)\) time where \(T\) is the number of threads. Can we apply FastTrack’s epoch optimizations to DC analysis?

DC rule (a). Tracking DC rule (a) requires \(O(T \times L)\) time (lines 16–18 and 24–26) for each access inside of critical sections on \(L\) locks, where \(T\) is the thread count; we find that

\(^{11}\) WCP analysis provides the same property (WCP rule (b) [47]) at a somewhat lower cost because it can maintain per-lock queues for each thread, instead of each pair of threads, as a consequence of WCP composing with HB [47].
many of our evaluated real programs have a high proportion of accesses executing inside one or more critical sections (Table 6.3 on page 149). Furthermore, maintaining $L_{m,x}^r$ and $L_{m,x}^w$ entails storing information for lock–variable pairs, requiring indirect metadata lookups (e.g., an implementation can use per-lock hash tables keyed by variables or per-variable hash tables keyed by locks). Note that applying FastTrack’s epoch optimizations to last-access metadata alone would not improve the cost of computing rule (a), and $L_{m,x}^r$ and $L_{m,x}^w$ cannot be represented using epochs.

The information needed to track DC rule (a) and rule (b) is unique to predictive analysis and poses a significant challenging component to optimize that is absent in HB analysis. Section 6.3.2 details optimizing tracking conflicting critical sections and its synergy with epoch and ownership optimizations to provide additional performance improvements.

### 6.3 SmartTrack: Optimized Predictive Analysis

This section introduces SmartTrack, a set of analysis optimizations applicable to predictive analyses:

1. *Epoch and ownership optimizations* from prior work optimizing HB analysis [31, 35, 100], applied to DC and WCP analyses (Section 6.3.1). To our knowledge, SmartTrack is the first application of these optimizations to predictive analysis.

2. A *conflicting critical section (CCS) optimization* that targets predictive analyses (Section 6.3.2). This novel optimization represents the chapter’s most significant intellectual contribution.
6.3.1 Epoch and Ownership Optimizations

In 2009, Flanagan and Freund introduced *epoch optimizations* to HB analysis, realized in the *FastTrack* algorithm [31]. The core idea is that HB analysis only needs to track the latest write to a variable $x$, and in some cases only needs to track the latest read to $x$, to detect the first race soundly. So FastTrack replaces the use of a vector clock with an *epoch*, $c@t$, to represent the latest write or read, where $c$ is an integer clock value and $t$ is a thread ID. The lightweight epoch representation is sufficient for detecting the first race soundly since if an access races with a prior write *not* represented by the last-write epoch, then it must also race with the last write (similarly for reads in some cases). Intuitively, if there’s no race with the last write and the current access, then either (1) the current access does not race with any earlier writes or (2) the last write races with an earlier write (which would have been detected earlier). A similar argument applies to reads.

It is straightforward to adapt FastTrack’s epoch optimizations for last-access metadata updates to *predictive* analysis, because changes to $R_x$ and $W_x$’s representations do not affect the logic for detecting CCSs. Though the core idea is straightforward to adapt to last-access metadata updates for predictive analysis, it is not a simple application to computing DC rule (a) or (b). So we first introduce SmartTrack in part before presenting the entirety of SmartTrack.

We base the SmartTrack algorithm on a FastTrack variant called *FastTrack-Ownership* (FTO) [100], which provides invariants that help make SmartTrack elegant and efficient. We explain FTO in the context of applying it to DC analysis.

The first step in developing SmartTrack is to apply FTO to DC, WCP, and WDC analyses. Algorithm 11 shows *FTO-DC*, which extends unoptimized DC analysis (Algorithm 8) to apply FTO’s optimizations to maintain last read and write metadata ($R_x$ and $W_x$) and
check for DC-races. Making similar changes to unoptimized WCP and WDC analysis is straightforward. (Removing lines 2 and 6–10 from Algorithm 11 yields FTO-WDC.)

As mentioned above, an epoch is a scalar $c@t$, where $c$ is a non-negative integer, and the leading bits represent $t$, a thread ID. $\perp$ denotes an uninitialized epoch. For simplicity of exposition, for the rest of the chapter, we redefine vector clocks to map to epochs instead of integers: $C : Tid \mapsto Epoch$, and redefine $C_1 \sqsubseteq C_2$ and $C_1 \sqcup C_2$ in terms of epochs. The notation $e \preceq C$ checks whether an epoch $e = c@t$ is ordered before a vector clock $C$, and evaluates to $c \leq c'$ where $c'@t = C(t)$.

FTO-DC analysis modifies the metadata used by unoptimized DC analysis (Algorithm 8) in the following ways:

- $W_x$ is an epoch representing the latest write to $x$.
- $R_x$ is either an epoch or a vector clock representing the latest reads and write to $x$.

Initially, every $R_x$ and $W_x$ is $\perp$.

Compared with unoptimized DC analysis (Algorithm 8), FTO-DC in Algorithm 11 significantly changes the maintenance and checking of $R_x$ and $W_x$, using a set of increasingly complex cases, which we describe in turn. Note, cases are presented as short circuit checks. Once a case succeeds the event updates and returns. Otherwise, a failed case implies a lack of ordering understood by the next case.

**Same-epoch cases.** At a write (or read) to $x$ by $t$, if $t$ has already written (or read or written) $x$ since the last synchronization event, then the access is effectively redundant (it cannot introduce a race or change last-access metadata). FTO-DC checks these cases by comparing the current thread’s epoch with $R_x$ or $W_x$, shown in the [READ SAME EPOCH], [SHARED SAME EPOCH], and [WRITE SAME EPOCH] cases in Algorithm 11.
Algorithm 11  
FTO-DC (FTO-based DC analysis)

1: procedure ACQUIRE(t, m)
2:   foreach t' ≠ t do Acq_{m,t}(t).Enque(C_t)
3:   C_t(t) ← C_t(t) + 1  \quad \text{\small \textgreater Supports same-epoch checks}
4: end procedure
5: procedure RELEASE(t, m)
6:   foreach t' ≠ t do
7:      while Acq_{m,t}(t').Front() ⊆ C_t do
8:         Acq_{m,t}(t').Deque()
9:      C_t ← C_t ∪ Rel_{m,t}(t').Deque()
10:   foreach x ∈ R_m do L'_{m,x} ← L'_{m,x} ∪ C_t
11:   foreach x ∈ W_m do L''_{m,x} ← L''_{m,x} ∪ C_t
12:   R_m ← W_m ← ∅
13:   C_t(t) ← C_t(t) + 1
14: end procedure
15: procedure WRITE(t, x)
16:   if W_x = C_t(t) then return \quad \text{[WRITE SAME EPOCH]}
17:   foreach m ∈ HeldLocks(t) do
18:      C_t ← C_t ∪ (L'_{m,x} ∪ L''_{m,x})
19:      W_m ← W_m ∪ \{x\}
20:      R_m ← R_m ∪ \{x\}
21:   if R_x = c@t then skip \quad \text{[WRITE OWNED]}
22:   else if R_x = c@u then \quad \text{[WRITE EXCLUSIVE]}
23:      check R_x ≤ C_t
24:   else \quad \text{[WRITE SHARED]}
25:      check R_x ⊆ C_t
26:      W_x ← R_x ← C_t(t)
27: end procedure
28: procedure READ(t, x)
29:   if R_x = C_t(t) then return \quad \text{[READ SAME EPOCH]}
30:   if R_x(t) = C_t(t) then return \quad \text{[SHARED SAME EPOCH]}
31:   foreach m ∈ HeldLocks(t) do
32:      C_t ← C_t ∪ L''_{m,x}
33:      R_m ← R_m ∪ \{x\}
34:   if R_x = c@t then \quad \text{[READ OWNED]}
35:      R_x ← C_t(t)
36:   else if R_x = c@u then \quad \text{[READ EXCLUSIVE]}
37:      if R_x ≤ C_t then
38:         R_x ← C_t(t)
39:      else \quad \text{[READ SHARE]}
40:         check W_x ≤ C_t
41:         R_x ← \{c@u, C_t(t)\}
42:      else if R_x(t) = c@t then \quad \text{[READ SHARED OWNED]}
43:         R_x(t) ← C_t(t)
44:      else \quad \text{[READ SHARED]}
45:         check W_x ≤ C_t
46:         R_x(t) ← C_t(t)
47: end procedure

133
The same-epoch check works because a thread increments its logical clock $C_t(t)$ at every synchronization operation. Although tracking DC (or WDC or WCP or HB) requires incrementing $C_t(t)$ only at release events, FTO-DC also increments $C_t(t)$ at acquire events to support correct same-epoch checks. Otherwise ACQUIRE and RELEASE are the same as in unoptimized DC analysis (Algorithm 8).

If a same-epoch case does not apply, then FTO-DC adds ordering from prior conflicting critical sections identical to Algorithm 8, before checking other FTO-DC cases. Note that at writes, FTO-DC updates $R_x$ as well as $W_x$ (line 27) and $R_m$ as well as $W_m$ (line 21) because $R_x$ and $L^r_{m,x}$ represent the last reads and writes.

**Owned cases.** At a read or write to $x$ by $t$, if $R_x$ represents a prior access by $t$ (i.e., $R_x = c@t$ or $R_x(t) \neq \perp$), then the current access cannot race with any prior accesses. The [READ OWNED], [READ SHARED OWNED], and [WRITE OWNED] cases skip the race check and proceed to update $R_x$ and/or $W_x$.

**Exclusive cases.** If an owned case does not apply and $R_x$ is an epoch, FTO-DC compares the current time with $R_x$. If the current access is a write, this comparison acts as a race check [WRITE EXCLUSIVE]. If the current access is a read, then the comparison determines whether $R_x$ can remain an epoch or must become a vector clock. If $R_x$ is DC ordered before the current access, then $R_x$ remains an epoch [READ EXCLUSIVE]. Otherwise, the algorithm checks for a write–read race by comparing the current access with $W_x$ and upgrades $R_x$ to a vector clock representing both the current read and prior read or write [READ SHARE].

**Shared cases.** If an owned case does not apply and $R_x$ is a vector clock, a “shared” case handles the access. Since $R_x$ may not be DC ordered before the current access, [READ
SHARED) checks for a race by comparing with $W_x$, while [WRITE SHARED] checks for a race by comparing with $R_x$ (comparing with $W_x$ is unnecessary since $W_x \preceq R_x$).

6.3.2 Optimizing Detecting Conflicting Critical Sections

While epoch and ownership optimizations improve the performance of predictive analyses, these optimizations do nothing to alleviate detecting conflicting critical sections (CCSs) to compute DC (or WCP or WDC) rule (a) (i.e., the second significant cost identified in Section 6.2) or rule (b) (for DC and WCP). Moreover, applying epoch optimizations directly to CCSs does not accurately track the DC relation.

Our insight for efficiently detecting CCSs is that, in most cases, an algorithm can maintain CCS metadata for a variable $x$ in the same way it maintains last-access metadata for $x$ (i.e., $R_x$ and $W_x$). SmartTrack detects CCSs using new analysis state $L^w_x$ and $L^r_x$, which mirror $W_x$ and $R_x$. $L^w_x$ represents critical sections containing the write represented by $W_x$; and $L^r_x$ represents critical sections containing the read or write represented by $R_x$ if $R_x$ is an epoch or $L^r_x$ represents a vector of critical sections containing the reads and/or writes represented by $R_x$ if $R_x$ is a vector clock. Representing CCSs in this manner leads to cheaper logic than prior algorithms for predictive analysis in the common case.

The idea is that if an access within a critical section conflicts with a prior access in a critical section on the same lock not represented by $L^w_x$ and $L^r_x$, then it must conflict with the last access within a critical section, represented by $L^w_x$ and $L^r_x$, or races with the last access. Though correct if all prior access’s critical sections are ordered before the current access, to precisely track DC, “ancillary” metadata is maintained for any CCS information lost when updating $L^w_x$ and $L^r_x$. 

135
Furthermore, the algorithm exploits the synergy between CCS and last-access metadata, often avoiding race checks due to detection of CCSs.

Algorithm 12 shows SmartTrack-DC, which applies the SmartTrack algorithm to DC analysis. Note that SmartTrack-DC builds on FTO-DC (Algorithm 11) by integrating CCS optimizations. Applying SmartTrack to both WCP and WDC analysis is analogous and straightforward. The rest of this section describes how SmartTrack works in the context of SmartTrack-DC.

**Analysis state.** SmartTrack introduces a new data type: the critical section (CS) list, which represents the logical times of releases of active critical sections at some time $c$ on thread $t$. A CS list has the following form:

$$\langle\langle C_1, m_1\rangle, \ldots, \langle C_n, m_n\rangle\rangle$$

where $m_1 \ldots m_n$ are locks held at time $c @ t$, in innermost to outermost order; and $C_1 \ldots C_n$ are references to (equivalently, shallow copies of) vector clocks representing the release time of each critical section at time $c @ t$, in innermost to outermost order. CS lists store references to vector clocks to enable deferred updating of $C_i$ until the release of $m_i$ occurs.

SmartTrack-DC maintains analysis state similar to Algorithm 11 with the following changes and additions:

- $H_t$ for each thread $t$, which is a current CS list for $t$;
- $L^w_x$ for each variable $x$ (replaces FTO-DC’s $L^w_{m,x}$), which is a CS list for the last write access to $x$;
- $L^r_x$ (replaces FTO-DC’s $L^r_{m,x}$) has a form dependent on $R_x$: 136
- \( L'_x \) is a CS list for the last read or write to \( x \) if \( R_x \) is an epoch;

- \( L'_x \) is a thread-indexed vector of CS lists (\( Tid \mapsto CS \) list) if \( R_x \) is a vector clock, with 
  \( L'_x(t) \) representing the CS list for the last read or write to \( x \) by \( t \);

- \( A^w_x \) and \( A^r_x \) (“ancillary” metadata) for each variable \( x \), which are vectors of maps from 
  locks to references to vector clocks (\( Tid \mapsto Lock \mapsto VC \)) that represent critical sections 
  containing accesses to \( x \) that are not necessarily captured by \( L^w_x \) and \( L^r_x \), respectively.

A straightforward addition to optimizing CCS detection, SmartTrack-DC makes the following change relative to FTO-DC:

- \( Acq_{m,t}(t') \) is now a queue of epochs.

Initially all CS lists are empty; \( A^w_x \) and \( A^r_x \) are empty maps.

**Maintaining CS lists.** SmartTrack-DC uses the same cases as FTO-DC (e.g., [READ SAME EPOCH] and [WRITE OWNED]). At each read or write to \( x \), SmartTrack-DC maintains 
CCS metadata in \( L^w_x \) and \( L^r_x \) that corresponds to last-access metadata in \( W_x \) and \( R_x \). At an 
access, the algorithm updates \( L^r_x \) and/or \( L^w_x \) to represent the current thread’s active critical 
sections.

SmartTrack-DC obtains the CS list representing the current thread’s active critical 
sections from \( H_t \), which the algorithm maintains at each acquire and release event. At an 
acquire, the algorithm prepends a new entry \( \langle C, m \rangle \) to \( H_t \) representing the new innermost 
critical section (lines 3–5). \( C \) is a reference to (i.e., shallow copy of) a newly allocated vector 
clock that represents the critical section’s release time, which is not yet known and will be 
updated at the release. In the meantime, another thread \( u \) may query whether \( t \)’s release of 
\( m \) is DC ordered before \( u \)’s current time (line 70; explained later). To ensure that this query
Algorithm 12 SmartTrack-DC (SmartTrack-based DC analysis)

1: procedure ACQUIRE(t, m)
2:   foreach t ′ ≠ t do Acq_{m,t′}(t).Enque(C_t(t))
3:   let C = reference to new vector clock
4:   C(t) ← ∞
5:   H_t ← ⟨C, m⟩ ⊕ H_t
6:   C_t(t) ← C_t(t) + 1
7: end procedure
8: procedure RELEASE(t, m)
9:   foreach t ′ ≠ t do
10:      while Acq_{m,t′}(t′).Front() ≤ C_t do
11:         Acq_{m,t′}(t′).Deque()
12:         C_t ← C_t ⊕ Rel_{m,t′}(t').Deque()
13:   end procedure
14:   let ⟨C, _⟩ = head(H_t)
15:   C ← C_t
16:   H_t ← rest(H_t)
17:   C_t(t) ← C_t(t) + 1
18: end procedure
19: procedure WRITE(t, x)
20:   if W_t = C_t(t) then return
21:      if A^n_t ≠ ∅ then
22:         foreach m ∈ HeldLocks(t) do
23:            C_t ← C_t ⊔ (∩_{u≠t} A^n_x(u)(m))
24:         end foreach
25:         foreach u ≠ t do A^n_x(u)(m) ← A^n_x(u)(m) ⊕ ∅
26:            A^n_t(t) ← A^n_t(t) ⊕ ∅
27:      else if R_t = c@t then skip
28:      else if R_t = c@u then
29:         let A = MULTICHECK(L^n_x(u), R_x)
30:         if A = ∅ then
31:            A^n_x(u) ← A
32:            A^n_x(u) ← MULTICHECK(L^n_x(u), ⊥)
33:         else
34:            foreach u ≠ t do
35:               let A = MULTICHECK(L^n_x(u), R_x(u))
36:               if A ≠ ∅ then
37:                  A^n_x(u) ← A
38:                  A^n_x(u) ← MULTICHECK(L^n_x(u), ⊥)
39:               end if
40:          end foreach
41:        end if
42:   end if
43:   W_t ← R_t ← C_t(t)
44: end procedure
Algorithm 12 [Continued]  SmartTrack-DC (SmartTrack-based DC analysis)

41: procedure READ(t, x)                        [Read Same Epoch]
42:     if Rx = Ct then return                  [Shared Same Epoch]
43:     if Rx(t) = Ct then return
44:         if A_w(t) ≠ ∅ then
45:             foreach m ∈ HeldLocks(t) do
46:                 Ct ← Ct ∪ (⎡\text{u} \neq t \rceil A_w(u)(m)⎤)
47:             if Rx = c@t then
48:                 L_rx ← H_t
49:                 Rx ← Ct(t)
50:         else if Rx = c@u then
51:             c′@u = \begin{cases} C′(u) & \text{s.t. } \langle C′, − \rangle = \text{tail}(L'_x) \text{ if } L'_x \neq \langle\rangle \\ Rx & \text{otherwise} \end{cases}
52:                 if c′@u ≤ Ct then
53:                     L'_x ← H_t
54:                     Rx ← Ct(t)
55:             else
56:                 MULTICHECK(L'_x, tid(W_x), W_x)
57:                 L'_x ← \{L'_x, H_t\}
58:                 Rx ← \{c@u, Ct(t)\}
59:         else if Rx(t) = c@t then
60:             L'_x(t) ← H_t
61:             Rx(t) ← Ct(t)
62:     else
63:         MULTICHECK(L'_x, tid(W_x), W_x)
64:             L'_x(t) ← H_t
65:             Rx(t) ← Ct(t)
66:     end procedure
67: procedure MULTICHECK(L, u, a)
68:     let A = ∅ 
69:     foreach (C, m) in L in tail-to-head order do
70:         if C(u) ≤ Ct then return A
71:             if m ∈ heldby(t) then
72:                 Ct ← Ct ∪ C
73:             return A
74:         A(m) ← C
75:     check a ≤ Ct
76:     return A
77: end procedure
returns false before $t$’s release of $m$, the algorithm initializes $C(t)$ to $\infty$ (line 4). When the release of $m$ happens, the algorithm removes the first element $\langle C, m \rangle$ of $H_t$, representing the critical section on $m$, and updates the vector clock referenced by $C$ with the release time (lines 14–16).

**Checking for CCSs and races.** At a read or write that may conflict with prior access(es), SmartTrack-DC combines the CCS check with the race check. To perform this combined check, the algorithm calls the helper function `MULTICHECK`. `MULTICHECK` traverses a CS list in reverse (outermost-to-innermost) order, looking for a prior critical section that is ordered to the current access or that conflicts with one of the current access’s held locks (lines 69–74). If a critical section matches, it subsumes checking for inner critical sections or a DC-race, so `MULTICHECK` returns. If no critical section matches, `MULTICHECK` performs the race check (line 75).

Figure 6.1(a) shows an example execution with arrows representing DC ordering. At Thread 1’s wr($x$), SmartTrack-DC sets $L_x^w$ to $\langle \langle C_n, n \rangle, \langle C_m, m \rangle, \langle C_p, p \rangle \rangle$, where $C_\ast$ are references to vector clocks such that $C_\ast(t_1) = \infty$ ($t_1$ represents Thread 1). At Thread 1’s rel($n$) and rel($m$), the algorithm sets $C_n$ and $C_m$, respectively, to the current time.

At Thread 2’s rd($x$), SmartTrack-DC takes the [READ SHARE] case because `MULTICHECK` detects $C_p$ (rel($p$)’s future time) is not DC ordered before the current time. (Later, in the context of Figure 6.1(b), we motivate why SmartTrack-DC must take the [READ SHARE] case here although FTO-DC would take the [READ EXCLUSIVE] case.) The [READ SHARE] case calls `MULTICHECK`, which first compares $C_p$ to the current time and then checks if Thread 2 holds $p$, which both correctly fail because Thread 1 has not released $p$ yet. `MULTICHECK` then compares $C_m$, the next lock in tail-to-head order, to the current time,
Figure 6.1: Example executions used by the text to illustrate how SmartTrack-DC works. All arrows show DC ordering. \( \text{sync}(o) \) represents the sequence \( \text{acq}(o); \text{rd}(o\text{Var}); \text{wr}(o\text{Var}); \text{rel}(o) \).
which fails since Thread 1’s \( \text{rel}(m) \) is not (yet) DC ordered to the current time, and then checks if \( m \) is held, which succeeds, so the algorithm adds DC ordering from \( \text{rel}(m) \) to \( \text{rd}(x) \), and \text{MULTICHECK} returns. Finally, the [\text{READ SHARE}] case sets \( L^x \) to a vector representing the prior and current critical sections containing \( x \), and \( R_x \) to a vector representing the prior and current accesses to \( x \).

At Thread 3’s \( \text{wr}(x) \), SmartTrack-DC takes the [\text{WRITE SHARED}] case. The algorithm first calls \text{MULTICHECK} to check ordering with Thread 1’s access; \text{MULTICHECK} detects no ordering from \( C_p (\text{rel}(p)’s \text{time}) \) to the current time, but detects the conflicting critical sections on \( p \), so it adds ordering from \( \text{rel}(p) \) to the current access. Next, the algorithm calls \text{MULTICHECK} to check ordering with Thread 2’s access, which succeeds immediately after detecting ordering from Thread 2’s \( \text{rel}(m) \) to the current access (due to the \text{sync(o)} events). Finally, the algorithm sets \( L^x_c \) and \( L^x_w \) each to a CS list representing Thread 3’s active critical section on \( p \), and \( R_x \) and \( W_x \) to Thread 3’s current epoch.

**SmartTrack’s [\text{READ SHARE}] behavior.** Optimizing CCSs presents a non-trival difference between SmartTrack-DC (FTO + CCS) and FTO-DC in that SmartTrack-DC takes the [\text{READ SHARE}] case in some situations when FTO-DC would take the [\text{READ EXCLUSIVE}] case. As mentioned above, for the execution in Figure 6.1(a), SmartTrack-DC takes [\text{READ SHARE}] at Thread 2’s \( \text{rd}(x) \), although FTO would take [\text{READ EXCLUSIVE}]. SmartTrack-DC takes the [\text{READ SHARE}] case because Thread 2’s \( \text{rd}(x) \) is not ordered after all of the last access’s critical sections, and taking the [\text{READ EXCLUSIVE}] case would lose information about Thread 1’s critical section on \( \text{rel}(p) \) containing \( x \).

Figure 6.1(b) shows an execution that motivates why this behavior is necessary. If SmartTrack-DC were to take the [\text{READ EXCLUSIVE}] case at Thread 2’s \( \text{rd}(x) \), then the
algorithm would lose information about Thread 1’s \( \text{rd}(x) \) being inside of the critical section on \( m \). As a result, SmartTrack-DC would miss adding ordering from Thread 1’s \( \text{rel}(m) \) to Thread 3’s \( \text{wr}(x) \) (dotted edge), leading to unsound tracking of DC and potentially reporting races that are not DC-races. SmartTrack-DC thus takes [**Read Share**] in situations like Thread 2’s \( \text{rd}(x) \) when the prior access’s critical sections (represented by the CS list \( L'_x \)) are not all ordered before the current access.

**Using “ancillary” metadata.** Owing in part to its [**Read Share**] behavior, SmartTrack-DC does not lose any needed CCS information at *reads*. Essentially, the algorithm only overwrites a CS list in \( L'_x \) for a thread \( t \) because of a new access to \( x \) by \( t \), which naturally subsumes critical sections of the prior access to \( x \) by \( t \).

However, SmartTrack-DC can lose needed CCS information at *writes* to \( x \), by overwriting information about critical sections in \( L'_x \) and \( L'_w \) that are not ordered before the current write. Figures 6.1(c) and 6.1(d) show two executions in which this situation occurs. In each execution, at Thread 2’s \( \text{wr}(x) \), SmartTrack-DC updates \( L'_x \) and \( L'_w \) to \( \langle \rangle \) (representing the access’s lack of active critical sections)—which loses information about Thread 1’s critical section on \( m \) containing an access to \( x \). As a result, in each execution, when Thread 3 accesses \( x \), SmartTrack-DC cannot use \( L'_x \) or \( L'_w \) to detect the ordering from Thread 1’s \( \text{rel}(m) \) to the current access.

To ensure sound tracking of DC, SmartTrack-DC uses the “ancillary” metadata \( A'_x \) and \( A'_w \) to track CCS information lost from \( L'_x \) and \( L'_w \) at writes to \( x \). \( A'_x(t)(m) \) and \( A'_w(t)(m) \) each represent the release time of a critical section on \( m \) by \( t \) containing a read or write \( (A'_r) \) or write \( (A'_w) \) to \( x \). **MULTICHECK** computes a “residual” map \( A \) of critical sections that are not ordered to the current access (line 74), which SmartTrack-DC assigns to \( A'_x \) or \( A'_w \). At
a write or read not handled by a same-epoch case, if $A_r^r$ or $A_w^w$, respectively, is non-empty, the analysis adds ordering for CCSs represented in $A_r^r$ (lines 21–25) or $A_w^w$ (lines 44–46), respectively.

In essence, SmartTrack-DC uses per-variable CCS metadata ($L_r^r$ and $L_w^w$) that mimics last-access metadata ($R_x$ and $W_x$) when feasible, and otherwise falls back to CCS metadata ($A_r^r$ and $A_w^w$) analogous to non-SmartTrack DC algorithms (i.e., $L_{m,x}^r$ and $L_{m,x}^w$ in Algorithms 8 and 11). SmartTrack-DC’s performance improvement over FTO-DC relies on $A_r^r$ and $A_w^w$ being empty in most cases.

**Optimizing $Acq_{m,i}(t')$.** A final optimization that we include as part of SmartTrack-DC is to change $Acq_{m,i}(t')$ from a vector clock (used in FTO-DC) to an epoch. This optimization to how DC rule (b) is computed is correct because an epoch is sufficient for checking if ordering has been established from an $acq(m)$ on $t'$ to a $rel(m)$ on $t$, since SmartTrack-DC increments $C_t(t)$ after every acquire operation.

Note that SmartTrack is directly applicable to WDC and WCP analyses. Removing lines 2 and 9–13 from Algorithm 12 yields SmartTrack-WDC.

### 6.3.3 Performance Cost of Soundness for DC Analysis

A final significant cost of DC analysis is supporting the *vindication* algorithm (Chapter 5) that checks whether a DC-race is a predictable race. Building a constraint graph during DC analysis, even with optimizations such as event node merging, can add significant time and space overhead.

Instead of constructing an event graph, an implementation of DC analysis can either (1) report all DC-races, which are almost always predictable races in practice, or (2) use multithreaded deterministic replay techniques to replay a recorded execution that detected a
(previously unknown) DC-race, using DC analysis that constructs an event graph during the replayed execution. Recent record & replay approaches add very low (3%) run-time overhead to record an execution [53, 62]. Replay failure caused by undetected races [51] is a non-issue since DC analysis detects all races.

6.4 Evaluation

This section evaluates the effectiveness of our predictive analysis optimizations.

6.4.1 Implementation

Table 6.1 presents the evaluated analysis implementations, categorized by analysis type (row headings) and optimization level (column headings). Each cell in the table (e.g., FTO-WDC) is an analysis that represents the application of an algorithm (FTO) to an analysis type (WDC analysis).

For the unoptimized analyses (Unopt columns), we used the publicly available Vindicator implementation [79],12 which implements vector-clock-based HB, WCP, and DC analyses, and the VINDICATE-RACE algorithm for checking DC-races. We extended DC analysis to

\[ \text{https://github.com/PLaSSticity/Vindicator} \]

<table>
<thead>
<tr>
<th></th>
<th>Unopt w/G</th>
<th>Unopt (w/o G)</th>
<th>Epochs</th>
<th>+ Ownership</th>
<th>+ CS optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>HB</td>
<td>N/A</td>
<td>Unopt-HB</td>
<td>FastTrack2 [35]</td>
<td>FTO-HB [100]</td>
<td>N/A</td>
</tr>
<tr>
<td>WCP</td>
<td>N/A</td>
<td>Unopt-WCP</td>
<td>—</td>
<td>FTO-WCP</td>
<td>SmartTrack-WCP</td>
</tr>
<tr>
<td></td>
<td>Unopt-DC w/G [79]</td>
<td>Unopt-DC (Algo. 8)</td>
<td>—</td>
<td>FTO-DC (Algo. 11)</td>
<td>SmartTrack-DC</td>
</tr>
<tr>
<td>WDC</td>
<td>Unopt-WDC w/G</td>
<td>Unopt-WDC</td>
<td>—</td>
<td>FTO-WDC</td>
<td>SmartTrack-WDC</td>
</tr>
</tbody>
</table>

Table 6.1: Evaluated analyses. Optimizations increase from left to right, and relations weaken from top to bottom.
implement WDC analysis. Vindicator is build on RoadRunner version 0.5, as discussed in Section 2.4 (page 20). We implemented the optimized analyses (+ Ownership and + CS optimizations) based on RoadRunner’s default FastTrack2 analysis [35].

**Same-epoch cases.** The Unopt-* analysis implementations perform a [SHARED SAME EPOCH]-like check at reads and writes (not shown in Algorithm 8). For correctness, the unoptimized predictive analysis implementations (Unopt-{WCP, DC, WDC}) increment $C_i(t)$ at acquires as well as releases, just as for the optimized predictive analyses.

**Handling races.** As described in Section 2.4, the analyses handle executions up to the first race and then continues normally after detecting a race.

**Analysis metadata.** Each analysis processes events correctly in parallel by using fine-grained synchronization on analysis metadata. An analysis can forgo synchronization for an access if a same-epoch check succeeds. To synchronize this lock-free check correctly, the read and write epochs in all analyses are volatile variables.

### 6.4.2 Methodology

Each analysis is evaluated separately using the DaCapo benchmarks discussed in Section 2.5 (page 22).

Each reported performance result, race count, or frequency statistic for an evaluated program is the arithmetic mean of 10 trials. We measure execution time as wall-clock time within the benchmarked harness of the evaluated program, and memory usage as the maximum resident set size during execution according to the GNU time program. We
Table 6.2: Frequencies of non-same-epoch reads and writes for SmartTrack-WDC, for each evaluated program.

<table>
<thead>
<tr>
<th>Program</th>
<th>Event Type</th>
<th>Total</th>
<th>Owned</th>
<th>Unowned</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Excl</td>
<td>Shared</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Excl</td>
<td>Share</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Share</td>
<td></td>
</tr>
<tr>
<td>avrora</td>
<td>Read</td>
<td>94 M</td>
<td>42.2%</td>
<td>53.9%</td>
</tr>
<tr>
<td></td>
<td>Write</td>
<td>44 M</td>
<td>98%</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.8%</td>
<td>1.1%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.94%</td>
<td></td>
</tr>
<tr>
<td>batik</td>
<td>Read</td>
<td>3.2 M</td>
<td>100%</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>Write</td>
<td>2.4 M</td>
<td>100%</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.0069%</td>
<td>&lt;0.001%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>N/A</td>
<td>&lt;0.001%</td>
</tr>
<tr>
<td>h2</td>
<td>Read</td>
<td>250 M</td>
<td>82.7%</td>
<td>8.7%</td>
</tr>
<tr>
<td></td>
<td>Write</td>
<td>46 M</td>
<td>98.9%</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7.6%</td>
<td>0.25%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.85%</td>
<td></td>
</tr>
<tr>
<td>jython</td>
<td>Read</td>
<td>110 M</td>
<td>95.2%</td>
<td>4.8%</td>
</tr>
<tr>
<td></td>
<td>Write</td>
<td>28 M</td>
<td>100%</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&lt;0.001%</td>
<td>&lt;0.001%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>N/A</td>
<td>&lt;0.001%</td>
</tr>
<tr>
<td>luindex</td>
<td>Read</td>
<td>27 M</td>
<td>100%</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>Write</td>
<td>13 M</td>
<td>100%</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&lt;0.001%</td>
<td>&lt;0.001%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>N/A</td>
<td>&lt;0.001%</td>
</tr>
<tr>
<td>lusearch</td>
<td>Read</td>
<td>110 M</td>
<td>96.1%</td>
<td>3.9%</td>
</tr>
<tr>
<td></td>
<td>Write</td>
<td>28 M</td>
<td>100%</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&lt;0.001%</td>
<td>&lt;0.001%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>N/A</td>
<td>0.0011%</td>
</tr>
<tr>
<td>pmd</td>
<td>Read</td>
<td>7.4 M</td>
<td>98%</td>
<td>1.6%</td>
</tr>
<tr>
<td></td>
<td>Write</td>
<td>0.54 M</td>
<td>98.5%</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.12%</td>
<td>0.16%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.15%</td>
<td></td>
</tr>
<tr>
<td>sunflow</td>
<td>Read</td>
<td>2.5 M</td>
<td>3.9%</td>
<td>56%</td>
</tr>
<tr>
<td></td>
<td>Write</td>
<td>0.96 M</td>
<td>100%</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&lt;0.001%</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.016%</td>
<td></td>
</tr>
<tr>
<td>tomcat</td>
<td>Read</td>
<td>5.0 M</td>
<td>36.5%</td>
<td>47.6%</td>
</tr>
<tr>
<td></td>
<td>Write</td>
<td>3.9 M</td>
<td>39.6%</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5.6%</td>
<td>7.2%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3.2%</td>
<td></td>
</tr>
<tr>
<td>xalan</td>
<td>Read</td>
<td>190 M</td>
<td>82.3%</td>
<td>17.6%</td>
</tr>
<tr>
<td></td>
<td>Write</td>
<td>40 M</td>
<td>89.1%</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.012%</td>
<td>0.020%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.039%</td>
<td></td>
</tr>
</tbody>
</table>

measure time, memory, and races within the same runs, and frequency statistics in separate runs that enable collecting statistics.

For simplicity, we do not include results with 95% confidence intervals but were evaluated to verify that differences in reported results involve overlapping confidence intervals.

6.4.3 Run-Time Characteristics

Table 6.2 reports frequencies of each FTO case for SmartTrack-WDC analysis, averaged over the 10 trials. The Total column counts the non-same-epoch reads and writes, i.e., all read and write events that do not take [READ SAME EPOCH], [SHARED SAME EPOCH], or [WRITE SAME EPOCH] cases. Each value in the remaining columns represents, for a specific read or write case, the percentage of the total non-same-epoch reads or writes, respectively.
Table 6.3 shows run-time characteristics relevant to the analyses. The \#Thr column shows the total number of threads created and, in parentheses, the maximum number of active threads at any time. Events are the total executed program events (All) and non-same-epoch accesses (NSEAs).

The Locks held at NSEAs columns report percentages of non-same-epoch accesses holding at least one, two, or three locks, respectively. These counts are important because non-SmartTrack predictive analyses perform substantial work per held lock at non-same-epoch accesses. While all programs generally benefit from epoch and ownership optimizations, only programs that perform many accesses holding one or more locks benefit substantially from the CCS optimization. Notably, h2, luindex, and xalan have the highest average locks held per access. Unsurprisingly, these programs have the highest FTO-based predictive analysis overhead and benefit the most from SmartTrack’s optimizations (Section 6.4.5).

The “Ancillary” metadata columns report percentages of non-same-epoch accesses that succeed the ancillary metadata Check (lines 21 and 44 in Algorithm 12) and Use ancillary metadata at least once to add critical section ordering (lines 23 and 46 in Algorithm 12). These counts are important since SmartTrack-based predictive analyses require ancillary metadata to soundly track the WCP, DC, and WDC relations at each program variable even when the ancillary metadata is rarely or never used, which is true for most benchmarks. Programs that perform few if any accesses holding one or more locks obtain trivial benefits from the CCS optimizations and in some cases the ancillary metadata incurs nontrivial performance overheads.
Table 6.3: Run-time characteristics of the evaluated programs. “NSEAs” are non-same-epoch accesses.

<table>
<thead>
<tr>
<th>Program</th>
<th>#Thr</th>
<th>Size (LoC)</th>
<th>Events All</th>
<th>NSEAs ≥ 1</th>
<th>≥ 2</th>
<th>≥ 3</th>
<th>“Ancillary” metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>avrora</td>
<td>7 (7)</td>
<td>69 K</td>
<td>1,400M 140M</td>
<td>5.9% &lt;0.1%</td>
<td>0</td>
<td>6.8%</td>
<td>0</td>
</tr>
<tr>
<td>batik</td>
<td>7 (2)</td>
<td>188 K</td>
<td>160M 5.8M</td>
<td>46.1% &lt;0.1%</td>
<td>0.1%</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>h2</td>
<td>10 (9)</td>
<td>116 K</td>
<td>3,800M 300M</td>
<td>82.8% 80.1%</td>
<td>0.17%</td>
<td>0.46%</td>
<td>&lt;0.001%</td>
</tr>
<tr>
<td>jython</td>
<td>2 (2)</td>
<td>212 K</td>
<td>730M 170M</td>
<td>3.8% 0.23%</td>
<td>&lt;0.1%</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>luindex</td>
<td>3 (3)</td>
<td>126 K</td>
<td>400M 41M</td>
<td>25.8% 25.4%</td>
<td>0.17%</td>
<td>0.46%</td>
<td>&lt;0.001%</td>
</tr>
<tr>
<td>lusearch</td>
<td>10 (10)</td>
<td>126 K</td>
<td>1,400M 140M</td>
<td>3.8% 0.39%</td>
<td>&lt;0.1%</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pmd</td>
<td>9 (9)</td>
<td>61 K</td>
<td>210M 8.0M</td>
<td>1.1% 0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sunflow</td>
<td>17 (17)</td>
<td>22 K</td>
<td>9,700M 3.5M</td>
<td>0.78% &lt;0.1%</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>tomcat</td>
<td>35 (35)</td>
<td>159 K</td>
<td>44M 9.7M</td>
<td>13.1% 8.0%</td>
<td>3.9%</td>
<td>0.13%</td>
<td>&lt;0.001%</td>
</tr>
<tr>
<td>xalan</td>
<td>9 (9)</td>
<td>176 K</td>
<td>630M 240M</td>
<td>99.9% 99.7%</td>
<td>1.1%</td>
<td>6.5%</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.4: Run-time and memory performance for FastTrack-based HB analyses and for unoptimized DC and WDC analyses, relative to uninstrumented execution.

<table>
<thead>
<tr>
<th>Program</th>
<th>HB FT2</th>
<th>Unopt-DC w/ G</th>
<th>Unopt-DC w/o G</th>
<th>Unopt-WDC w/ G</th>
<th>Unopt-WDC w/o G</th>
<th>HB FT2</th>
<th>Unopt-DC w/ G</th>
<th>Unopt-DC w/o G</th>
<th>Unopt-WDC w/ G</th>
<th>Unopt-WDC w/o G</th>
</tr>
</thead>
<tbody>
<tr>
<td>avrora</td>
<td>4.0× 3.9×</td>
<td>24× 21×</td>
<td>22× 18×</td>
<td>4.1× 4.1×</td>
<td>72× 42×</td>
<td>72× 37×</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>batik</td>
<td>4.1× 4.0×</td>
<td>11× 10×</td>
<td>11× 10×</td>
<td>4.9× 4.9×</td>
<td>46× 43×</td>
<td>44× 42×</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>h2</td>
<td>8.3× 8.3×</td>
<td>79× 85×</td>
<td>80× 89×</td>
<td>3.1× 3.1×</td>
<td>59× 57×</td>
<td>57× 65×</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jython</td>
<td>7.8× 7.7×</td>
<td>31× 25×</td>
<td>27× 22×</td>
<td>7.0× 7.0×</td>
<td>32× 18×</td>
<td>25× 16×</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>luindex</td>
<td>6.9× 6.9×</td>
<td>41× 36×</td>
<td>39× 36×</td>
<td>4.4× 4.4×</td>
<td>68× 53×</td>
<td>69× 53×</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lusearch</td>
<td>9.8× 9.8×</td>
<td>28× 26×</td>
<td>26× 29×</td>
<td>9.3× 9.3×</td>
<td>15× 13×</td>
<td>15× 13×</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pmd</td>
<td>6.3× 6.4×</td>
<td>17× 16×</td>
<td>15× 15×</td>
<td>2.9× 2.9×</td>
<td>15× 14×</td>
<td>14× 14×</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sunflow</td>
<td>15× 14×</td>
<td>74× 70×</td>
<td>74× 69×</td>
<td>8.4× 8.4×</td>
<td>43× 39×</td>
<td>41× 38×</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tomcat</td>
<td>4.2× 4.0×</td>
<td>15× 17×</td>
<td>9.8× 12×</td>
<td>2.7× 2.7×</td>
<td>28× 29×</td>
<td>21× 25×</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xalan</td>
<td>4.0× 4.0×</td>
<td>54× 45×</td>
<td>48× 39×</td>
<td>6.5× 6.5×</td>
<td>65× 59×</td>
<td>62× 59×</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>geomean</td>
<td>6.4× 6.3×</td>
<td>31× 28×</td>
<td>28× 27×</td>
<td>4.9× 4.9×</td>
<td>38× 32×</td>
<td>36× 31×</td>
<td>Run time</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 149
6.4.4 Comparing Baselines

Table 6.4 shows results that help determine whether we are using valid baselines compared with prior work. The left side of the table reports run times, with slowdown factors relative to uninstrumented execution (execution without any analysis instrumentation), and the right side reports memory usage, with usage factors relative to uninstrumented execution.

**FastTrack comparison.** The HB columns report the performance of two variants of the FastTrack algorithm. FT2 is our implementation of the FastTrack2 algorithm [35], based closely on RoadRunner’s implementation of FastTrack2, which is the default FastTrack tool in RoadRunner. The main difference between FT2 and RoadRunner’s FastTrack2 is in their handling of detected races. RoadRunner’s FastTrack2 does not update last-access metadata at read events that detect a race (for unknown reasons); it does not perform analysis on future accesses to a variable after it detects a race on the variable; and it limits the number of races it counts for by class field and array type. In contrast, our FT2 updates last-access metadata after every event even if it detects a race; it does not stop performing analysis on any events; and it counts every race.

FTO is our implementation of FTO-HB analysis, implemented in the same RoadRunner tool as FT2. Overall FTO-HB performs quite similarly to FT2. The rest of the chapter’s results compare against FTO-HB as the representative from the FastTrack family of HB analyses.

**Unoptimized analyses.** Table 6.4’s Unopt-* columns compare the performance of unoptimized DC and WDC analyses, with and without support for vindication. To support verifying that DC-races (or WDC-races) are (true) predictable races, DC (WDC) analysis
can build a constraint graph $G$ during the analysis. \textit{Unopt-DC w/G} represents the cost incurred by prior work to detect DC-races and be able to check them after execution. It also represents the cost that would be incurred by a replayed execution that builds $G$ in order to verify DC-races detected by a recorded run that used SmartTrack-DC analysis or some other DC analysis that does not build $G$ (Section 6.3.3). Likewise, \textit{Unopt-WDC w/G} shows the cost of a replayed execution checking WDC-races.

\textit{Unopt-DC w/o G} represents the cost incurred by prior work to detect DC-races without being able to check them, which is still useful because few if any DC-races are false positives in practice, and a second replayed run can optionally check DC-races. Likewise, \textit{Unopt-WDC w/o G} shows the cost of detecting WDC-races without being able to check them. The rest of the results compare our optimized analyses against unoptimized analyses that do not build a constraint graph (\textit{Unopt-* w/o G}).

The results show that the costs of unoptimized predictive analyses are high, whether or not they build a constraint graph, compared with existing optimized non-predictive (HB) analyses. The rest of the results evaluate whether our optimizations help to bridge this performance gap.
Table 6.6: Run time, relative to uninstrumented execution, of various analyses for each evaluated program.

<table>
<thead>
<tr>
<th></th>
<th>Unopt-</th>
<th>FTO-</th>
<th>ST-</th>
<th>Unopt-</th>
<th>FTO-</th>
<th>ST-</th>
<th>Unopt-</th>
<th>FTO-</th>
<th>ST-</th>
<th>Unopt-</th>
<th>FTO-</th>
<th>ST-</th>
</tr>
</thead>
<tbody>
<tr>
<td>HB</td>
<td>17 x</td>
<td>3.9 x</td>
<td>N/A</td>
<td>7.8 x</td>
<td>4.0 x</td>
<td>N/A</td>
<td>31 x</td>
<td>8.3 x</td>
<td>N/A</td>
<td>22 x</td>
<td>7.7 x</td>
<td>N/A</td>
</tr>
<tr>
<td>WCP</td>
<td>23 x</td>
<td>7.8 x</td>
<td>5.9 x</td>
<td>12 x</td>
<td>6.9 x</td>
<td>4.3 x</td>
<td>83 x</td>
<td>55 x</td>
<td>11 x</td>
<td>33 x</td>
<td>9.9 x</td>
<td>11 x</td>
</tr>
<tr>
<td>DC</td>
<td>21 x</td>
<td>8.0 x</td>
<td>6.4 x</td>
<td>10 x</td>
<td>6.9 x</td>
<td>4.2 x</td>
<td>85 x</td>
<td>55 x</td>
<td>10 x</td>
<td>25 x</td>
<td>10 x</td>
<td>10 x</td>
</tr>
<tr>
<td>WDC</td>
<td>18 x</td>
<td>6.1 x</td>
<td>4.5 x</td>
<td>10 x</td>
<td>6.8 x</td>
<td>4.1 x</td>
<td>89 x</td>
<td>52 x</td>
<td>9.5 x</td>
<td>22 x</td>
<td>8.2 x</td>
<td>8.2 x</td>
</tr>
<tr>
<td>avora</td>
<td>6.0 x</td>
<td>5.7 x</td>
<td>N/A</td>
<td>4.7 x</td>
<td>5.4 x</td>
<td>N/A</td>
<td>19 x</td>
<td>5.6 x</td>
<td>N/A</td>
<td>19 x</td>
<td>5.5 x</td>
<td>N/A</td>
</tr>
<tr>
<td>batik</td>
<td>4.6 x</td>
<td>4.8 x</td>
<td>N/A</td>
<td>6.1 x</td>
<td>6.3 x</td>
<td>N/A</td>
<td>20 x</td>
<td>5.2 x</td>
<td>N/A</td>
<td>20 x</td>
<td>5.1 x</td>
<td>N/A</td>
</tr>
<tr>
<td>h2</td>
<td>4.3 x</td>
<td>4.5 x</td>
<td>N/A</td>
<td>6.0 x</td>
<td>6.2 x</td>
<td>N/A</td>
<td>21 x</td>
<td>5.0 x</td>
<td>N/A</td>
<td>21 x</td>
<td>5.0 x</td>
<td>N/A</td>
</tr>
<tr>
<td>jython</td>
<td>3.5 x</td>
<td>3.6 x</td>
<td>N/A</td>
<td>4.8 x</td>
<td>4.9 x</td>
<td>N/A</td>
<td>19 x</td>
<td>4.6 x</td>
<td>N/A</td>
<td>19 x</td>
<td>4.6 x</td>
<td>N/A</td>
</tr>
<tr>
<td>sunflow</td>
<td>3.5 x</td>
<td>3.5 x</td>
<td>N/A</td>
<td>3.0 x</td>
<td>3.0 x</td>
<td>N/A</td>
<td>20 x</td>
<td>3.0 x</td>
<td>N/A</td>
<td>20 x</td>
<td>3.0 x</td>
<td>N/A</td>
</tr>
<tr>
<td>tomcat</td>
<td>3.5 x</td>
<td>3.5 x</td>
<td>N/A</td>
<td>3.0 x</td>
<td>3.0 x</td>
<td>N/A</td>
<td>21 x</td>
<td>3.0 x</td>
<td>N/A</td>
<td>21 x</td>
<td>3.0 x</td>
<td>N/A</td>
</tr>
<tr>
<td>xalan</td>
<td>3.5 x</td>
<td>3.5 x</td>
<td>N/A</td>
<td>3.0 x</td>
<td>3.0 x</td>
<td>N/A</td>
<td>22 x</td>
<td>3.0 x</td>
<td>N/A</td>
<td>22 x</td>
<td>3.0 x</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 6.7: Memory usage, relative to uninstrumented execution, of various analyses for each evaluated program.

<table>
<thead>
<tr>
<th></th>
<th>Unopt-</th>
<th>FTO-</th>
<th>ST-</th>
<th>Unopt-</th>
<th>FTO-</th>
<th>ST-</th>
<th>Unopt-</th>
<th>FTO-</th>
<th>ST-</th>
<th>Unopt-</th>
<th>FTO-</th>
<th>ST-</th>
</tr>
</thead>
<tbody>
<tr>
<td>HB</td>
<td>32 x</td>
<td>4.1 x</td>
<td>N/A</td>
<td>30 x</td>
<td>4.9 x</td>
<td>N/A</td>
<td>15 x</td>
<td>3.1 x</td>
<td>N/A</td>
<td>21 x</td>
<td>7.0 x</td>
<td>N/A</td>
</tr>
<tr>
<td>WCP</td>
<td>99 x</td>
<td>11 x</td>
<td>7.2 x</td>
<td>46 x</td>
<td>13 x</td>
<td>6.1 x</td>
<td>65 x</td>
<td>40 x</td>
<td>4.9 x</td>
<td>21 x</td>
<td>13 x</td>
<td>10 x</td>
</tr>
<tr>
<td>DC</td>
<td>42 x</td>
<td>11 x</td>
<td>7.2 x</td>
<td>43 x</td>
<td>13 x</td>
<td>5.7 x</td>
<td>57 x</td>
<td>39 x</td>
<td>4.9 x</td>
<td>18 x</td>
<td>12 x</td>
<td>11 x</td>
</tr>
<tr>
<td>WDC</td>
<td>37 x</td>
<td>8.3 x</td>
<td>4.7 x</td>
<td>42 x</td>
<td>13 x</td>
<td>5.7 x</td>
<td>65 x</td>
<td>40 x</td>
<td>4.4 x</td>
<td>16 x</td>
<td>8.7 x</td>
<td>7.5 x</td>
</tr>
<tr>
<td>avora</td>
<td>4.0 x</td>
<td>3.9 x</td>
<td>N/A</td>
<td>3.0 x</td>
<td>3.9 x</td>
<td>N/A</td>
<td>20 x</td>
<td>3.0 x</td>
<td>N/A</td>
<td>20 x</td>
<td>3.0 x</td>
<td>N/A</td>
</tr>
<tr>
<td>batik</td>
<td>4.0 x</td>
<td>3.9 x</td>
<td>N/A</td>
<td>3.0 x</td>
<td>3.9 x</td>
<td>N/A</td>
<td>21 x</td>
<td>3.0 x</td>
<td>N/A</td>
<td>21 x</td>
<td>3.0 x</td>
<td>N/A</td>
</tr>
<tr>
<td>h2</td>
<td>4.0 x</td>
<td>3.9 x</td>
<td>N/A</td>
<td>3.0 x</td>
<td>3.9 x</td>
<td>N/A</td>
<td>22 x</td>
<td>3.0 x</td>
<td>N/A</td>
<td>22 x</td>
<td>3.0 x</td>
<td>N/A</td>
</tr>
<tr>
<td>jython</td>
<td>4.0 x</td>
<td>3.9 x</td>
<td>N/A</td>
<td>3.0 x</td>
<td>3.9 x</td>
<td>N/A</td>
<td>23 x</td>
<td>3.0 x</td>
<td>N/A</td>
<td>23 x</td>
<td>3.0 x</td>
<td>N/A</td>
</tr>
<tr>
<td>sunflow</td>
<td>4.0 x</td>
<td>3.9 x</td>
<td>N/A</td>
<td>3.0 x</td>
<td>3.9 x</td>
<td>N/A</td>
<td>24 x</td>
<td>3.0 x</td>
<td>N/A</td>
<td>24 x</td>
<td>3.0 x</td>
<td>N/A</td>
</tr>
<tr>
<td>tomcat</td>
<td>4.0 x</td>
<td>3.9 x</td>
<td>N/A</td>
<td>3.0 x</td>
<td>3.9 x</td>
<td>N/A</td>
<td>25 x</td>
<td>3.0 x</td>
<td>N/A</td>
<td>25 x</td>
<td>3.0 x</td>
<td>N/A</td>
</tr>
<tr>
<td>xalan</td>
<td>4.0 x</td>
<td>3.9 x</td>
<td>N/A</td>
<td>3.0 x</td>
<td>3.9 x</td>
<td>N/A</td>
<td>26 x</td>
<td>3.0 x</td>
<td>N/A</td>
<td>26 x</td>
<td>3.0 x</td>
<td>N/A</td>
</tr>
</tbody>
</table>
6.4.5 Run-Time and Memory Performance

This section evaluates the performance of our optimized analyses, compared with competing approaches from prior work. Tables 6.5, 6.6, and 6.7 show this chapter’s main results: run time and memory usage (relative to uninstrumented execution) of 11 analyses from Table 6.1. For example, a cell in column \( ST \) and row \( DC \) shows the performance of SmartTrack-DC analysis. Table 6.5 reports the geometric mean across all programs, and Tables 6.6 and 6.7 show separate results for each program.

The main takeaway is that SmartTrack’s optimizations are effective at improving the performance of all three predictive analyses substantially, achieving performance (notably run-time overhead) close to state-of-the-art HB analysis. On average across the programs, the FTO optimizations applied to predictive analyses result in a 2.2–2.6 \( \times \) speedup and 2.7–3.6 \( \times \) memory usage reduction over unoptimized analyses (Unopt-*), although FTO-based predictive analyses are still about twice as slow as FTO-HB on average. SmartTrack (which adds CCS optimizations to FTO) provides a 1.5–1.7 \( \times \) average speedup and 1.6–1.8 \( \times \) memory usage reduction over FTO-* analyses, showing that CCS optimizations eliminate most of the remaining costs FTO-based predictive analyses incur compared with FTO-HB. Overall, SmartTrack optimizations yield 3.3–4.1 \( \times \) average speedups and 4.2–6.3 \( \times \) memory usage reductions over unoptimized analyses, closing the performance gap compared with FTO-HB. Both FTO and CCS optimizations contribute proportionate improvements to achieve predictive analysis with performance close to that of state-of-the-art HB analysis.

HB analysis generally outperforms predictive analyses at each optimization level because it is the most straightforward analysis, eschewing the cost of computing CCSs by simply ordering all critical sections. Unopt-WCP performs worse than Unopt-DC due to the
Table 6.8: Average races reported by various analyses for each evaluated program (excluding batik and lusearch, for which all analyses report no races). In each cell, the first value is statically distinct races (i.e., distinct program locations) and the second value, in parentheses, is total dynamic races. As the text explains, significant differences between the algorithms (Unopt-, FTO-, ST-) are attributable to how the algorithms behave after detecting the first race.

<table>
<thead>
<tr>
<th></th>
<th>Unopt-</th>
<th>FTO-</th>
<th>ST-</th>
<th></th>
<th>Unopt-</th>
<th>FTO-</th>
<th>ST-</th>
<th></th>
<th>Unopt-</th>
<th>FTO-</th>
<th>ST-</th>
</tr>
</thead>
<tbody>
<tr>
<td>HB</td>
<td>6 (426,723)</td>
<td>6 (405,573)</td>
<td>N/A</td>
<td>13 (89,105)</td>
<td>13 (44,928)</td>
<td>N/A</td>
<td>21 (22)</td>
<td>24 (26)</td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WCP</td>
<td>6 (423,373)</td>
<td>6 (404,954)</td>
<td>6 (406,346)</td>
<td>13 (89,916)</td>
<td>13 (85,936)</td>
<td>13 (66,303)</td>
<td>22 (25)</td>
<td>19 (20)</td>
<td>25 (26)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DC</td>
<td>6 (443,731)</td>
<td>6 (404,498)</td>
<td>6 (406,902)</td>
<td>13 (91,074)</td>
<td>13 (86,383)</td>
<td>13 (70,557)</td>
<td>31 (35)</td>
<td>27 (30)</td>
<td>31 (34)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WDC</td>
<td>6 (440,519)</td>
<td>6 (406,375)</td>
<td>6 (408,624)</td>
<td>13 (91,092)</td>
<td>13 (86,657)</td>
<td>13 (50,618)</td>
<td>31 (35)</td>
<td>29 (31)</td>
<td>27 (30)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Avrora

<table>
<thead>
<tr>
<th></th>
<th>Unopt-</th>
<th>FTO-</th>
<th>ST-</th>
<th></th>
<th>Unopt-</th>
<th>FTO-</th>
<th>ST-</th>
<th></th>
<th>Unopt-</th>
<th>FTO-</th>
<th>ST-</th>
</tr>
</thead>
<tbody>
<tr>
<td>HB</td>
<td>1 (1)</td>
<td>1 (1)</td>
<td>N/A</td>
<td>7 (320)</td>
<td>18 (1,668)</td>
<td>N/A</td>
<td>6 (49)</td>
<td>6 (51)</td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WCP</td>
<td>1 (1)</td>
<td>1 (1)</td>
<td>1 (1)</td>
<td>6 (443)</td>
<td>18 (1,770)</td>
<td>18 (1,659)</td>
<td>18 (288)</td>
<td>18 (219)</td>
<td>19 (250)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DC</td>
<td>1 (1)</td>
<td>1 (1)</td>
<td>1 (1)</td>
<td>11 (1,623)</td>
<td>18 (3,511)</td>
<td>19 (3,835)</td>
<td>19 (734)</td>
<td>19 (406)</td>
<td>19 (406)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WDC</td>
<td>1 (1)</td>
<td>1 (1)</td>
<td>1 (1)</td>
<td>10 (1,555)</td>
<td>18 (3,671)</td>
<td>18 (3,537)</td>
<td>19 (677)</td>
<td>19 (402)</td>
<td>19 (413)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

H2

<table>
<thead>
<tr>
<th></th>
<th>Unopt-</th>
<th>FTO-</th>
<th>ST-</th>
<th></th>
<th>Unopt-</th>
<th>FTO-</th>
<th>ST-</th>
<th></th>
<th>Unopt-</th>
<th>FTO-</th>
<th>ST-</th>
</tr>
</thead>
<tbody>
<tr>
<td>HB</td>
<td>637 (1,782,554)</td>
<td>601 (1,489,958)</td>
<td>N/A</td>
<td>8 (2,324)</td>
<td>8 (2,584)</td>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WCP</td>
<td>591 (1,635,042)</td>
<td>620 (1,465,152)</td>
<td>612 (1,499,621)</td>
<td>64 (5,307,857)</td>
<td>43 (3,589,223)</td>
<td>50 (3,578,681)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DC</td>
<td>629 (1,559,537)</td>
<td>623 (1,438,653)</td>
<td>613 (1,493,252)</td>
<td>78 (7,444,803)</td>
<td>53 (3,965,177)</td>
<td>52 (3,984,338)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WDC</td>
<td>656 (1,833,028)</td>
<td>622 (1,450,638)</td>
<td>608 (1,485,273)</td>
<td>78 (7,437,373)</td>
<td>53 (3,892,399)</td>
<td>53 (3,987,647)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Jython

<table>
<thead>
<tr>
<th></th>
<th>Unopt-</th>
<th>FTO-</th>
<th>ST-</th>
<th></th>
<th>Unopt-</th>
<th>FTO-</th>
<th>ST-</th>
<th></th>
<th>Unopt-</th>
<th>FTO-</th>
<th>ST-</th>
</tr>
</thead>
<tbody>
<tr>
<td>HB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WCP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WDC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Luindex

<table>
<thead>
<tr>
<th></th>
<th>Unopt-</th>
<th>FTO-</th>
<th>ST-</th>
<th></th>
<th>Unopt-</th>
<th>FTO-</th>
<th>ST-</th>
<th></th>
<th>Unopt-</th>
<th>FTO-</th>
<th>ST-</th>
</tr>
</thead>
<tbody>
<tr>
<td>HB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WCP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WDC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pmd

<table>
<thead>
<tr>
<th></th>
<th>Unopt-</th>
<th>FTO-</th>
<th>ST-</th>
<th></th>
<th>Unopt-</th>
<th>FTO-</th>
<th>ST-</th>
<th></th>
<th>Unopt-</th>
<th>FTO-</th>
<th>ST-</th>
</tr>
</thead>
<tbody>
<tr>
<td>HB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WCP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WDC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sunflow

<table>
<thead>
<tr>
<th></th>
<th>Unopt-</th>
<th>FTO-</th>
<th>ST-</th>
<th></th>
<th>Unopt-</th>
<th>FTO-</th>
<th>ST-</th>
<th></th>
<th>Unopt-</th>
<th>FTO-</th>
<th>ST-</th>
</tr>
</thead>
<tbody>
<tr>
<td>HB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WCP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WDC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tomcat

<table>
<thead>
<tr>
<th></th>
<th>Unopt-</th>
<th>FTO-</th>
<th>ST-</th>
<th></th>
<th>Unopt-</th>
<th>FTO-</th>
<th>ST-</th>
<th></th>
<th>Unopt-</th>
<th>FTO-</th>
<th>ST-</th>
</tr>
</thead>
<tbody>
<tr>
<td>HB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WCP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WDC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Xalan

additional cost of computing HB (needed to compute WCP). FTO-WCP and SmartTrack-WCP reduce this analysis cost significantly. At the same time, DC rule (b) is somewhat more complex to compute than WCP rule (b) (Section 6.3.3). These two effects cancel out on average, leading to little or no average performance difference between FTO-WCP and FTO-DC and between SmartTrack-WCP and SmartTrack-DC. WDC analysis eliminates computing rule (b), achieving better performance than DC analysis at all optimization levels.

SmartTrack thus enables running three kinds of predictive analysis, each offering a different coverage–soundness tradeoff, with performance approaching HB analysis. Next, we evaluate the predictive power of these analyses.
6.4.6 Predictable Race Coverage

Although our evaluation focuses on the performance of our optimizations, and prior work and Chapter 5 has established that WCP and DC analyses, respectively, detect more races than HB analysis [47, 79], we have also evaluated how many races each analysis detects.

Table 6.8 reports how many races each analysis finds. For each cell, the second value (in parentheses) is total dynamic races reported, and the first value is statically distinct races. Two dynamic races detected at the same static program location are the same statically unique race.

Comparing relations. In general, the results confirm that weaker relations find more races than stronger relations. However, although the analyses get progressively more powerful from top to bottom (e.g., every DC-race is a WDC-race), this relationship does not always hold empirically for two reasons. First, run-to-run variation naturally affects repeatability. For simplicity of presentation, tables with 95% confidence intervals for these results are not shown but were evaluated to verify that many of the differences involve overlapping confidence intervals. Second, analyses have different performance characteristics that may affect an execution’s timing and memory access interleaving, leading to different races occurring.

The results for this chapter often report many more races, especially dynamic races, than Chapter 5’s results. These differences occur because Chapter 5’s evaluation used default RoadRunner behavior that stops performing analysis for a field after 100 dynamic races detected on the field, whereas this chapter’s evaluation disables that behavior.
The results do show that despite using a weaker relation than DC analysis, WDC analysis does not on average report more races than DC analysis, which suggests that WDC analysis’s optimization does not lead to false races in practice. In separate experiments with the Unopt-{DC, WDC} w/G analyses, VINDICATE RACE successfully vindicates every DC- and WDC-race detected across 10 trials, confirming every dynamic DC- and WDC-race is a true predictable race.

Comparing optimizations. For each relation, the different algorithms (Unopt-, FTO-, ST-) often report comparable race counts, but sometimes the counts differ significantly. These differences—particularly between unoptimized (Unopt-) and optimized (FTO-, ST-) race counts—occur because of run-to-run variation and performance characteristics, but primarily for a third reason: the different optimization levels have different behavior after they detect the first race, affecting race counts by using different metadata (e.g., epochs vs. vector clocks) to update racing accesses and detect future races.

Thus for each relation, the differences between the algorithms (Unopt-, FTO-, ST-) are not a reflection of race detection effectiveness across optimizations. Any extra races detected by one algorithm are likely to be related to each other (e.g., extra races involve accesses to the same data structure as accesses in races reported by all algorithms, or extra races may be dependent on races reported by all algorithms), and thus not be of much use to programmers. Rather, the race differences serve to show that the proposed optimizations and our implementations of them lead to reasonable race detection results.

6.4.7 Evaluation Summary

As the results show, unoptimized prior work’s WCP, DC, and WDC analyses are expensive, particularly for programs that frequently access variables in critical sections.
The SmartTrack-optimized WCP, DC, and WDC analyses improve run-time and memory performance by several times on average, achieving performance comparable to HB analysis.

SmartTrack’s optimizations are effective across predictive analyses and demonstrates practical use during in-house testing as a possible substitute to HB analysis. Sound WCP analysis detects fewer races than other predictive analyses and, in its unoptimized form, has the highest overhead. SmartTrack-WCP provides performance on par with HB analysis and other predictive analyses. At the other end of the coverage–soundness tradeoff, WDC analysis has the potential to detect the most false races (although in practice it detects only true predictable races), and it has the lowest overhead among predictive analyses. We emphasize that SmartTrack-DC and SmartTrack-WDC analyses do not perform vindication. They instead report DC- and WDC-races without vindication; or they can use record & replay techniques to replay an execution and collect enough information to perform vindication.

Evaluating analysis effectiveness across multiple executions is admittedly beyond the scope of this dissertation. A key challenge is the plethora of different options for choosing multiple executions (cf. Chapter 7). Simply re-executing a program natively is likely to expose a similar set of HB-races, while an advanced race-exposing technique could potentially expose HB-races effectively, but likely at a higher cost. Despite this, the results show that predictive analyses can be practical data race detectors that are competitive with standard highly optimized HB data race detectors.

6.5 Contribution and Impact

The novel SmartTrack algorithm improves run time and memory usage performance of predictive analyses that closes the gap in performance between predictive analysis and highly optimized HB analysis. To provide practical high-coverage predictive analysis,
SmartTrack introduces optimizations that apply to a family of predictive analyses offering different coverage–soundness tradeoffs. This dissertation is the first to apply epoch and ownership optimizations to predictive analyses which yield lightweight management of access metadata. Targeting partial-order-based predictive analyses, SmartTrack optimizes the unique component of detecting conflicting critical sections. As the evaluation shows, SmartTrack improves the performance of predictive analyses significantly over prior work to be competitive with highly optimized HB analysis while detecting more races than HB analysis. Thus SmartTrack achieves the goal of developing a practical high-coverage predictive analysis with trade-offs between race coverage and soundness guarantees.

Existing partial-order-based predictive analyses detect more races than industry standard HB analysis but are impractical and still miss races knowable from the observed execution. SmartTrack accomplishes strengthening race detection capability beyond existing predictive analyses and improves the run time and memory usage to close the performance gap with highly optimized HB analysis while providing tradeoffs to soundness guarantees established by prior predictive analyses. These results suggest the potential for using SmartTrack-based predictive race detection analysis regularly during in-house testing.
Chapter 7: Related Work

This chapter covers prior work that has proposed an extensive variety of approaches for detecting data races.

Static analysis. Static race detection can detect all feasible data races across every execution [27, 66, 67, 76, 99], but it is inherently unsound (reports false races). In practice, existing techniques report thousands of false races (e.g., [5, 50]). Developers generally avoid unsound analyses (analyses that permit false data races) because each reported race—whether true or false—takes substantial time to investigate [4, 14, 31, 36, 59, 68].

Some prior work uses static analysis to drive dynamic analysis, but the high rate of false positives limits its benefits [21, 26, 50, 77, 98].

Dynamic analysis. Dynamic analysis techniques developed to detect and report the existence of data races within a concurrent program largely use a lockset-based approach [81] or a happens-before (HB) approach [31].

Lockset-based approaches use a locking discipline to determine if a common lock is held by two conflicting accesses which indicates proper synchronization [21, 23, 69, 70, 81, 97]. The locking discipline over simplifies proper synchronization by reducing it to common locks held while disregarding other forms of proper synchronization (such as relying on a properly synchronized flag as control flow synchronization). Although lockset analysis can
predict data races in other executions, it is inherently unsound (reports false positives) since not all violations of a locking discipline are data races (e.g., accesses ordered by fork, join, or notify-wait synchronization).

As Chapter 3 mentioned, the HB components of Raptor are similar to prior work’s Goldilocks analysis [26]. Goldilocks and Raptor use per-variable “locksets” or sets, respectively, to track the HB relation (and in Raptor’s case, the CP relation) soundly and completely [26]. In contrast, lockset analysis is a different kind of analysis that detects violations of a locking discipline that requires conflicting memory accesses to hold a common lock.

**HB analysis** detects conflicting events unordered by the HB relation [26, 31, 49, 75], a partial order of events in the execution. The partial order is usually a subset of the total order in the execution that allows conflicting accesses to be considered consecutive, indicating the presence of a data race in a valid reordered execution. Several other non-predictive analyses detect a similar set of races, based on conflicting regions or forcing conflicting events to happen simultaneously [5, 6, 24, 28, 83, 95].

**Hybrid approaches** utilize aspects of both HB and lockset analyses for performance or accuracy reasons [45, 70, 75, 102], but it inherently cannot soundly predict data races in other executions.

**Predictive analysis.** Sound predictive analysis detects data races that are possible in an execution other than the observed execution [20, 40, 41, 47, 54, 80, 85, 92]. Some predictive approaches detect races beyond those knowable from an observed dynamic execution alone, by encoding static control-flow constraints in addition to dynamic constraints [40, 41, 85]. Most existing sound predictive analyses cannot scale to full program executions; they instead
analyze bounded windows of execution (e.g., 500-10,000 events), so they cannot predict data races between accesses that are “far apart” in the observed execution [20,40,41,54,80,85,92]. An exception is Kini et al.’s recent weak-causally-precedes (WCP) analysis, which can analyze whole program executions. However, WCP analysis misses races that are knowable from a dynamic execution, not only in theory but also for real programs, as Section 2.3.3 shows.

Schedule exploration. Some approaches explore multiple thread interleaving schedules, either systematically or based on heuristics for exposing new behaviors [16,18,29,38,65,83]. Approaches that guide thread schedules [16,18,29,65,83], by randomization or information about a data race, increase the chance for conflicting accesses to execute consecutively. Similar approaches perturb an execution by pausing threads to consecutively execute possible conflicting accesses which exposes data races [5,28,45]. Maximal causality reduction uses predictive analysis to systematically explore all legal, distinct schedules [39,42]. Schedule exploration is complementary with predictive analysis, which enables finding more races in each explored schedule.

Production-time dynamic analysis. Data races manifest nondeterministically under specific thread interleavings, program inputs, and execution environments so that they may stay hidden even for extensively tested programs [94]. This nondeterminism can require tens or hundreds of runs or more to manifest a race [104] and takes weeks to reproduce, diagnose, and fix in production systems [36,55]. Unsurprisingly, data race detectors do not see much use in practice, both in testing and production environments [59].

A production-time analysis finds data races that occur in production settings, using sampling to trade coverage for performance [5,14,28,45,59,89,103] or introducing custom
hardware support [22, 74, 82, 101, 104]. Detecting data races in production is orthogonal to testing-time analysis, which can use more time and space to find races before deployment.

**Alternatives to detecting data races.** Researchers have introduced language, type, and system support for avoiding data races or providing well-defined behavior for them [1, 3, 8, 15, 30, 56, 60, 61, 63, 71, 78, 82, 84, 90, 91, 93]. Existing solutions have significant drawbacks that have limited their adoption, such as requiring writing code in new languages or adding type annotations, impacting production-time performance, or relying on custom hardware.

**Prioritizing data races.** Prior work seeks to expose erroneous *behaviors* due to data races, to prioritize races that are demonstrably harmful [17, 19, 28, 32, 44, 68, 83]. However, as researchers have argued convincingly, *all* data races are problematic because they lead to ill-defined semantics [2, 10, 11, 13, 58, 88]. In any case, prioritizing data races is complementary to our work, which tries to detect as many (true) data races as possible.
Chapter 8: Conclusion

This chapter summarizes the contributions presented in this dissertation and formulates ideas for future work.

8.1 Contributions

The challenge in writing and diagnosing shared-memory programs requires efficient tools that expose hard-to-detect data races with competitive performance to industry standard race detectors. Existing partial-order-based predictive analyses successfully expand data race coverage over industry standard HB analysis but remain incomplete and impractical for developers to trust their application reliably. This dissertation provides the solution of a practical high-coverage sound predictive analysis that strengthens race detection capability and improves run time and memory usage beyond existing predictive analyses.

This dissertation’s contributions focus on maximizing race detection capability. By weakening the order of events more than the weakest known partial orders through the elimination of the overly conservative composition with synchronization order our new relations and analyses detect more race than state-of-the-art predictive race detection. Our new online analysis for an inherently recursive partial order indirectly improves race coverage by analyzing long executions prior work is unable to. While our contributions detect all knowable races from an observed execution, we simultaneously verify each detected race to provide
the same soundness guarantees as prior predictive analyses and existing industry standard
detectors, thus strictly improving race detection capability. This dissertation presented a
body of work that improved data race coverage over existing predictive analyses.

The other focus of this dissertation is improving the practicality of predictive analyses.
Our contributions identify and optimize the main costs of existing predictive analyses being
the use of vector clocks for both last access metadata, and the metadata used to track
necessary critical section ordering. Our optimizations are the first to apply epoch and
ownership optimizations and novel conflicting critical section optimizations to a family
of predictive analyses. Our results show that our optimized predictive analyses improve
performance significantly over prior work and are competitive with widely used HB analysis,
thus achieving practical high-coverage predictive analysis.

The body of work presented in this dissertation is our solution to providing effective
predictive analyses for developers writing and diagnosing efficient shared-memory programs.

8.2 Impact

We believe that all data races are errors and we demonstrate the existence of and a need
to detect more data races than existing predictive analyses and industry standard detectors.
The body of work introduced in this dissertation achieves the goal of designing a practical
high-coverage sound predictive analysis that yields performance on par with the latest highly
optimized HB race detectors. This dissertation undertakes the challenge of rethinking how
to detect data races during testing and makes a strong case for predictive analysis being the
prevailing approach for race detection.

The focus on progressing race detection capability of this dissertation detects all pre-
dictable races in real program executions and thus answers an open question of just how
many predictable races exist in real programs. The DC and WDC relations that strengthen race detection capability opens an opportunity for further weakening and redesign of how to detect a data race. These contributions show that hard-to-detect data races still exist in programs and can be detected using proficient analyses from a single execution.

In an effort to match the soundness guarantees of existing predictive analyses and industry standard detectors, our contributions support the aggressive weakening of the ordering between events while maintaining reporting of only true data races. Future work can further improve data race coverage of predictive analyses and other techniques that rely on dynamic analysis by building on the insights of weakening the observed order discussed in this dissertation.

The goal of improving the run time and memory usage performance of this dissertation identifies the main costs of partial-order-based predictive analyses. The optimizations developed for predictive analyses to be competitive with highly optimized HB analysis show the utilization of existing work to and practical application of current and future work in predictive analysis. Insights into optimizing essential analysis metadata discussed in this dissertation benefit future work to yield competitive predictive analyses. Redefining a data race or how to reason about execution reordering allows for more efficient dynamic data race detection techniques.

This dissertation and future development continue to strengthen race detection capability and improve run time and memory usage performance of data race detectors that will provide developers with practical tools to guarantee reliable concurrent software. As a result, developers will be more capable in writing and diagnosing efficient shared-memory programs using robust predictive analysis during testing to find as many data races knowable from dynamic executions.
Bibliography


[98] Christoph von Praun and Thomas R. Gross. Static Conflict Analysis for Multi-
Threaded Object-Oriented Programs. In *ACM Conference on Programming Language

[99] Jan Wen Young, Ranjit Jhala, and Sorin Lerner. RELAY: Static Race Detection on
Millions of Lines of Code. In *European Software Engineering Conference and ACM
SIGSOFT International Symposium on Foundations of Software Engineering*, pages

[100] Benjamin P. Wood, Man Cao, Michael D. Bond, and Dan Grossman. Instrumentation
Bias for Dynamic Data Race Detection. *Proceedings of the ACM on Programming

[101] Benjamin P. Wood, Luis Ceze, and Dan Grossman. Low-Level Detection of Language-
Level Data Races with LARD. In *ACM International Conference on Architectural

[102] Yuan Yu, Tom Rodeheffer, and Wei Chen. RaceTrack: Efficient Detection of Data
Race Conditions via Adaptive Tracking. In *ACM Symposium on Operating Systems

[103] Tong Zhang, Changhee Jung, and Dongyoon Lee. ProRace: Practical Data Race
Detection for Production Use. In *ACM International Conference on Architectural

[104] Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. HARD: Hardware-Assisted
Lockset-based Race Detection. In *International Symposium on High-Performance