ASSERTIONS FOR DEBUGGING
PARALLEL PROGRAMS

DANIEL SCHWARTZ-NARBONNE

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
ELECTRICAL ENGINEERING
ADVISER: PROFESSOR SHARAD MALIK

JUNE 2013
Abstract

Software engineering is facing a crisis. For decades, computer programs doubled in speed every 18 months, enabling powerful new applications which have literally changed the way we communicate and think. However, the ever-increasing hardware speed that drove this exponential growth also led to an unsustainable increase in power consumption. Modern hardware attempts to side-step this problem by using many slow processors functioning in parallel. Unfortunately, our ability to develop and debug parallel software to take advantage of parallel hardware has not kept pace with the need. In order to reap the benefits offered by parallel computing, we require new tools.

In particular, a parallel program must execute correctly even in the presence of unpredictable thread interleavings. This interleaving makes it hard to write correct parallel programs, and also makes it hard to find bugs in incorrect parallel programs. A range of tools have been developed to help debug parallel programs, ranging from atomicity-violation and data-race detectors to model-checkers and theorem provers. One technique that has been successful for debugging sequential programs, but less effective for parallel programs, is running the program using assertion predicates provided by the developer.

These assertions allow programmers to specify and check their assumptions. In a multi-threaded program, the programmer’s assumptions include both the current state, and any actions (e.g., access to shared memory) that other, parallel executing threads might take. This dissertation introduces parallel assertions which allow programmers to express these assumptions for parallel programs using simple and intuitive syntax and semantics. I present an implementation and demonstrate its performance using PARSEC benchmarks. I discuss the challenges of evaluating assertions in the presence of weak memory models, and show how my work formalizing
the semantics under weak memory models leads to an optimization that gives a 2x speedup while allowing more bugs to be exposed.

I measure the effectiveness of parallel assertions using the University of Michigan Collection of Concurrency Bugs. In 14/17 cases, I was able to write a parallel assertion that would detect the cause of the real-world bug. My research suggests that parallel assertions are a powerful and practical debugging tool.
Acknowledgements

I would like to thank my adviser Professor Sharad Malik, who has been a mentor to me throughout my work towards my Ph.D. When I started at Princeton, I knew how to be a student; Sharad taught me how to be a researcher. Academically, he taught me to take care and pride in my work. On a personal level, I have witnessed again and again how deeply Sharad cares about his students as people. His quick intellect and uncanny ability to find the gaps in a scientific argument kept me on my toes intellectually; his encouragement for me to take up marathon running after my accident helped keep me on my toes physically as well.

I would like to thank my research collaborators on this project: David August, Feng Liu, Sharad Malik, Tarun Pondicherry, and Georg Weissenbacher. I couldn’t have done it without you.

I would also like to thank my readers, Margaret Martonosi and Georg Weissenbacher. Their careful edits made this dissertation clearer and more accurate. Thanks to David Walker and David Wentzlaff for agreeing to serve as examiners on my committee.

A university cannot run without a dedicated support staff. I would like to thank Lori Bailey, Colleen Conrad, Sarah McGovern, and Stacey Weber, for their tireless work making sure things run smoothly for the students.

My work has been supported by a number of organizations. I gratefully acknowledge Semiconductor Research Corporation (SRC), Microelectronics Advanced Research Corporation (MARCO) Focus Center Research Program’s (FCRP) Gigascale Systems Research Center (GSRC), and the Natural Sciences and Engineering Research Council of Canada (NSERC) for their financial support during my Ph.D. I would also like to thank Intel, who supported me through two summer internships at their research office in Haifa.
I would like to express my deepest appreciation to Bill Watterson for creating *Calvin and Hobbes*, one of the greatest comic strips of all time, and to Universal Uclick for generously allowing me to include his artwork in this dissertation.

A graduate school career is about much more than just research. I could never have made it through the endurance challenge that is a Ph.D. without the friendship of some amazing people. There are so many names to mention here that it is almost certain that I will forget someone important. If I have, please know that it is my famously bad memory for names, and mentally add your name to the list.

My research group has been a source of support both academically and personally. Yogesh Mahajan shepherded me through my first paper. Carven Chan was a valuable collaborator, as well as the best dessert chef I have ever met. Thanks to Arnab Sinha for lending me his beloved car for a summer, during which time it drove to Ontario, Vermont and New York, and survived a major hurricane. Divjyot Sethi (AKA DJ) was endlessly entertaining. Yavuz Yetim, Shuyuan Zhang, and I spent more hours than I can count in the server room trying to keep the SAT cluster running. Charlie Shucheng Zhu kept me looking good. Sunha Ahn always brightened the room with her cheerfulness and energy. Georg Weissenbacher was always there to help me with the heavy lifting, whether with a marker in front of the white board, or with a barbell at the gym. Matthias Schlaipfer was great to go to a game with, even if he cheered for the wrong hockey team. Nestan Tsiskaridze encouraged me to go to concerts in the chapel. Best of luck to Burchin Cakir and Pramod Subramanyan, who represent the future of the Malik group. I will miss you, and my C305 office, very much.

My graduate school friends kept me grounded. They encouraged me to work when I needed to work; they also encouraged me to play when I needed to play. Special thanks to the EE crew: Lorne Applebaum, Sushobhan Avasthi, Abhishek Bhattacharjee, Richard Cendejas, Stephanie Goldfarb (an honorary EE), Noah Jafferis, Kevin Loutherback, Andrea Nedic, Clinton Smith and Helen Yu. We are now scattered...
all over the country, but I look forward to meeting up and building another igloo someday. Thanks also to Katie Mack, who could always be relied on when I needed someone to go for ice cream. Kelly Matula and Carra Glatt, thank you for bridging the grad/undergrad divide, and being my friends. In addition to being a wonderful friend, Jeff Dwoskin also generously provided the \LaTeX template that was used to typeset this thesis.

A successful dissertation requires fuel for both the mind and body. I would like to thank the many people in Princeton who helped keep my life well-balanced. The members of the Centre for Information Technology Policy (CITP) who expanded my intellectual horizons, and made sure that I never had to travel far for a free lunch. The dedicated volunteers who organized cookie-time every Friday. The enthusiastic musicians of “Rise Up Singing”, who helped me forget about my troubles every Wednesday night, and the Murray-Dodge bakers who kept us supplied with fresh cookies as we sang. The “Angry Nerds” softball team, who gave me an excuse to drink beer on hot summer days. Da, the owner of Da’s Kitchen, set the standard of Thai cooking by which I judge other restaurants; I will miss both her delicious food, and her highly informative cooking lessons. I would like to thank the Princeton Fire Department for allowing me to fulfill my boyhood dream of becoming a fireman, particularly Lt. Tom Scherer, who found the time to help a new probie learn the ropes. I similarly owe a debt of gratitude to the fleet-footed runners of ASHA/AID, who convinced me to run a marathon with them, and even to wake up before the crack of dawn to train. The prayer leaders at Kesher, the staff of the Centre for Jewish Life (CJL) and the organizers of Zamru all helped me to regain my spiritual grounding every Friday night.

Although my Canadian friends may not have been with me physically in Princeton, they were there in spirit. Julieta Chan, Mike Degan, Iris Elliott, Bernie Fitzpatrick, Giselle Gos, Sarah Greene, Hiyam Hameed, Tanya Hauck, David Kennedy,
Yonah Medbh Lavery-Yisraeli, Ian Leroux, Caitlyn Paget, Richard Pollard, Rebecca Saari, Chris ERTW Sapiano, Anne Sealey, Stephen Shapiro, Allie Simmonds, Leo Sin, Jeanette Veber, and many others who I am sure I have forgotten on this list, your visits, phone calls, and long internet chats kept me connected and recharged.

My family has always been a source of support and inspiration for me. My mother Roslyn taught me never to be satisfied with anything less than the best that I could do, and spent countless hours helping me polish my writing. My father Guy showed by example how to be a true academic in the best sense of the word. My sisters are some of the most amazing people I know. Rachel is working on a thesis that is 10× cooler than my work will ever be (someday, I will figure out a way to use Woolly Mammoth teeth in my research as well). Melody is incredibly talented. Figure 2.8 is her work, and she contributed greatly to the graphic design of my slides. Heather taught me by example never to let a little thing like a broken bone slow you down. Chantel is so inquisitive, and I can’t wait to see what she grows up to do. My personal vote is for Engineer, but only time will tell ©. I would also like to thank Lila Castro, who was one of my guides through life.

I would also like to thank the soon-to-be members of my family, the Liebowitzes and the Wheatleys, who have welcomed me so warmly. Thank you particularly to Kit Wheatley for doing so much to organize the wedding while I frantically worked on this dissertation, and to Jon Liebowitz for his careful copy-editing.

I would like to dedicate this work to my fiancée Anna Liebowitz. You have made the last five years some of the happiest of my life. You were always there when I needed you, whether it was to care for me as I recovered from my accident, or to listen when I needed to talk about my day. Thank you for always encouraging me to finish, and for understanding when things went slowly. This is for you.
For Anna.
Contents

Abstract ................................................................. iii
Acknowledgements ..................................................... v
List of Figures .......................................................... xvi

1 Introduction .......................................................... 1
  1.1 Overview .......................................................... 1
  1.2 Structure of this thesis .......................................... 2
  1.3 Background ....................................................... 4
    1.3.1 The democratization of computing .................... 4
    1.3.2 The power wall ............................................ 8
    1.3.3 The concurrent programming challenge .............. 11
  1.4 How can computer science help? ............................ 14
  1.5 Goals of this thesis ............................................ 15

2 The \texttt{passert} assertion language .......................... 16
  2.1 Introduction .................................................... 16
  2.2 What are sequential assertions? ............................ 17
    2.2.1 Example of sequential assertion use ................. 18
  2.3 Sequential assertions work for sequential programs .... 18
    2.3.1 Sequential assertions are easy to add .............. 18
    2.3.2 Sequential assertions are expressive .............. 19
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3.3</td>
<td>Sequential assertions minimally affect code execution</td>
<td>19</td>
</tr>
<tr>
<td>2.4</td>
<td>Sequential assertions do not work for parallel programs</td>
<td>20</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Sequential assertions are not evaluated at the right times</td>
<td>20</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Sequential assertions cannot check the right properties</td>
<td>23</td>
</tr>
<tr>
<td>2.5</td>
<td>The Parallel Assertion Statement</td>
<td>26</td>
</tr>
<tr>
<td>2.5.1</td>
<td>When they check: assertion scope</td>
<td>27</td>
</tr>
<tr>
<td>2.5.2</td>
<td>What they check: assertion condition</td>
<td>29</td>
</tr>
<tr>
<td>2.5.3</td>
<td>Formal Grammar</td>
<td>29</td>
</tr>
<tr>
<td>2.6</td>
<td>Parallel assertions work for parallel programs</td>
<td>33</td>
</tr>
<tr>
<td>2.6.1</td>
<td>Interference</td>
<td>33</td>
</tr>
<tr>
<td>2.6.2</td>
<td>Statement Atomicity</td>
<td>34</td>
</tr>
<tr>
<td>2.6.3</td>
<td>Thread Safety</td>
<td>34</td>
</tr>
<tr>
<td>2.6.4</td>
<td>ABA Problem</td>
<td>35</td>
</tr>
<tr>
<td>2.7</td>
<td>Related Work</td>
<td>35</td>
</tr>
<tr>
<td>2.7.1</td>
<td>Languages for correct by construction parallel code</td>
<td>35</td>
</tr>
<tr>
<td>2.7.2</td>
<td>Specification languages for debugging parallel code</td>
<td>36</td>
</tr>
<tr>
<td>2.8</td>
<td>Chapter Summary</td>
<td>38</td>
</tr>
<tr>
<td>3</td>
<td>The Semantics of Parallel Assertions</td>
<td>40</td>
</tr>
<tr>
<td>3.1</td>
<td>Memory Ordering Models</td>
<td>42</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Why memory ordering models are necessary</td>
<td>42</td>
</tr>
<tr>
<td>3.1.2</td>
<td>The granularity of memory ordering models</td>
<td>43</td>
</tr>
<tr>
<td>3.1.3</td>
<td>How memory ordering models are characterized</td>
<td>44</td>
</tr>
<tr>
<td>3.1.4</td>
<td>Fences</td>
<td>48</td>
</tr>
<tr>
<td>3.2</td>
<td>C/C++ semantics</td>
<td>49</td>
</tr>
<tr>
<td>3.3</td>
<td>Sequentially Consistent Semantics</td>
<td>51</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Observable Program Execution Timeline</td>
<td>52</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Instrumented State</td>
<td>57</td>
</tr>
</tbody>
</table>
3.3.3 Instrumented State Transition Function .......................... 59
3.3.4 Parallel Assertion Scope ........................................... 60
3.3.5 Parallel Assertion Condition ........................................ 61
3.3.6 Checking Parallel Assertions ....................................... 62
3.3.7 Evaluation Example ................................................... 62
3.4 Relaxed Ordering Semantics ............................................ 64
3.4.1 Observing Program Executions ..................................... 64
3.4.2 Operational Semantics of Parallel Assertions ..................... 71
3.5 Response to assertion violation ....................................... 78
3.5.1 Fail-stop ............................................................... 78
3.5.2 Fail before exiting scope ............................................ 79
3.5.3 Continue past failure ................................................ 79
3.6 Pointers and memory locations ...................................... 80
3.7 Related Work ........................................................... 81
3.7.1 Formal processor models .......................................... 81
3.7.2 Formal language models .......................................... 81
3.7.3 Debugging relaxed memory model systems ...................... 82
3.8 Chapter Summary ....................................................... 82

4 Implementation .......................................................... 84
4.1 Fundamental Design Decisions ........................................ 84
4.1.1 Why a dynamic runtime checker? ................................. 85
4.1.2 Why software based? ................................................. 85
4.1.3 Why compiler based? ................................................. 86
4.1.4 Why a decoupled log? ................................................. 87
4.2 The basic structure of passert ......................................... 87
4.2.1 Annotate code with assertions .................................... 88
4.2.2 Front-end ............................................................. 88

xii
4.2.3 Back-end ......................................................... 93
4.2.4 Runtime Library ............................................. 94
4.3 Benchmarks ....................................................... 98
  4.3.1 PARSEC .................................................... 98
  4.3.2 Annotated PARSEC benchmarks ......................... 99
4.4 Optimizations and Results ..................................... 101
  4.4.1 Online Checker Optimization ............................ 101
  4.4.2 Filtering Optimization .................................. 103
  4.4.3 Time-stamping Mechanisms ............................. 107
4.5 Related Work ................................................... 115
4.6 Chapter Summary ............................................... 116

5 Real World Applicability ........................................ 118
  5.1 Benchmark selection ......................................... 119
    5.1.1 The Michigan Collection of Concurrency Bugs ........ 119
    5.1.2 Representativeness of the Michigan Collection ....... 120
  5.2 Study design .................................................. 120
    5.2.1 Characterization of assertion effectiveness ......... 120
  5.3 Data race bugs ............................................... 121
    5.3.1 Apache #21287 ......................................... 121
    5.3.2 Apache #25520 ......................................... 124
    5.3.3 Cherokee Bug 1 ......................................... 124
    5.3.4 MySQL #644 ............................................ 124
    5.3.5 MySQL #791 ............................................ 126
    5.3.6 MySQL #2011 ........................................... 129
    5.3.7 Parallel Assertion ....................................... 129
    5.3.8 MySQL #3596 ........................................... 130
    5.3.9 MySQL #12848 .......................................... 131
5.4 Atomicity Bugs .................................................. 132
  5.4.1 Aget Bug 2 ........................................... 132
  5.4.2 Apache #45605 ..................................... 133
  5.4.3 memcached #127 ................................... 134
  5.4.4 MySQL #169 ......................................... 136
  5.4.5 MySQL #12228 ...................................... 138
  5.4.6 Apache #21285 ...................................... 138
5.5 Order violation bugs ............................................. 142
  5.5.1 pbzip2 0.9.4 .......................................... 142
  5.5.2 Transmission 1.42 .................................... 143
5.6 Deadlock bugs .................................................. 145
  5.6.1 Aget Bug1 .............................................. 145
5.7 Chapter Summary ................................................. 145
  5.7.1 Possible language improvements ...................... 147
  5.7.2 User-defined granularity ................................ 147

6 Conclusions ....................................................... 149
  6.1 Future work .................................................. 149
    6.1.1 Improvements to PASSERT ......................... 149
    6.1.2 Alternative Implementations ...................... 152
    6.1.3 Support for other programming languages/models .. 153
    6.1.4 Better benchmarks .................................. 154
    6.1.5 Fault localization .................................. 155
    6.1.6 Usability studies .................................. 156
    6.1.7 Automatic assertion generation .................... 156
  6.2 Final thoughts on parallel assertions .................. 157
    6.2.1 Are parallel assertions simple to use? .......... 157
    6.2.2 Are parallel assertions powerful enough to capture real bugs? 158
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Power consumption of Intel chips, by year of introduction. (from [75])</td>
<td>9</td>
</tr>
<tr>
<td>1.2</td>
<td>Compiler optimization causes unexpected results on multi-threaded code</td>
<td>14</td>
</tr>
<tr>
<td>2.1</td>
<td>GNU libc implementation of <code>assert</code></td>
<td>17</td>
</tr>
<tr>
<td>2.2</td>
<td>A simple sequential assertion</td>
<td>18</td>
</tr>
<tr>
<td>2.3</td>
<td>Translating a multi-column code layout into a sequentially consistent</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>interleaved trace</td>
<td></td>
</tr>
<tr>
<td>2.4</td>
<td>Interference Problem</td>
<td>22</td>
</tr>
<tr>
<td>2.5</td>
<td>Multiple assertions cannot reliably detect the interference problem</td>
<td>23</td>
</tr>
<tr>
<td>2.6</td>
<td>Statement Atomicity Problem</td>
<td>24</td>
</tr>
<tr>
<td>2.7</td>
<td>Thread Safety Problem</td>
<td>25</td>
</tr>
<tr>
<td>2.8</td>
<td>The ABA Problem</td>
<td>26</td>
</tr>
<tr>
<td>2.9</td>
<td>Parallel assertion scope and condition</td>
<td>27</td>
</tr>
<tr>
<td>2.10</td>
<td>Nested parallel assertion scopes</td>
<td>28</td>
</tr>
<tr>
<td>2.11</td>
<td>Multiple dynamic parallel assertion scopes</td>
<td>29</td>
</tr>
<tr>
<td>2.12</td>
<td>Parallel Assertion Operators</td>
<td>30</td>
</tr>
<tr>
<td>2.13</td>
<td>Example assertion conditions</td>
<td>30</td>
</tr>
<tr>
<td>2.14</td>
<td>Use of access operators on complex data-structures</td>
<td>32</td>
</tr>
<tr>
<td>2.15</td>
<td>Formal parallel assertions grammar for C/C++</td>
<td>33</td>
</tr>
<tr>
<td>2.16</td>
<td>Assertion to detect interference bug</td>
<td>33</td>
</tr>
<tr>
<td>2.17</td>
<td>Assertion to detect statement atomicity bug</td>
<td>34</td>
</tr>
</tbody>
</table>
5.1 Simple example of buggy program ........................................ 122
5.2 Assertion to detect symptom of bug in Figure 5.1 ................. 122
5.3 Assertion to diagnose cause of bug in Figure 5.1 ................. 122
5.4 Expected vs. actual behaviour for apr-atomic-dec ............... 123
5.5 Apache Bug #21287 ............................................... 123
5.6 Parallel assertion to find Apache Bug #21287 .................... 123
5.7 Apache #25520 ................................................ 125
5.8 Cherokee bug1 .................................................. 126
5.9 MySQL #644 .................................................... 127
5.10 Use of foreach on MySQL #644 .................................. 128
5.11 MySQL #791 .................................................... 128
5.12 MySQL #2011 .................................................. 129
5.13 MySQL #3596 .................................................. 130
5.14 thruIf(cond) operator ............................................ 131
5.15 Parallel assertion for MySQL #3596 using thruIf construct .. 131
5.16 MySQL #12848 .................................................. 132
5.17 Aget Bug2 ..................................................... 133
5.18 Apache #45605 ................................................ 135
5.19 memcached #127 ............................................... 136
5.20 MySQL #169 .................................................... 137
5.21 MySQL #12228 ................................................ 139
5.22 Apache #21285 ................................................ 141
5.23 Dynamically scoped parallel assertion for Apache #21285 .... 143
5.24 pbzip2-0.9.4 .................................................. 144
5.25 Transmission 1.42 .............................................. 145
5.26 Aget Bug 1 ...................................................... 146
5.27 Summary of parallel assertion bug coverage ...................... 148
Chapter 1

Introduction

1.1 Overview

Software engineering is facing a crisis. For decades, computer programs doubled in speed every 18 months, enabling powerful new applications which have literally changed the way we communicate and think. However, the ever-increasing hardware speed that drove this exponential growth also led to an unsustainable increase in power consumption.

Modern hardware attempts to side-step this problem by using many slow processors functioning in parallel. Unfortunately, our ability to develop and debug parallel software to take advantage of this new parallel hardware has not kept pace with the need. In order to reap the benefits offered by parallel computing, we require new tools. This dissertation presents a new and powerful assertion language which I have developed to help ordinary programmers debug parallel programs.
1.2 Structure of this thesis

Much of the work in this thesis has been previously published at scientific conferences, as described in Appendix A. Related work is discussed in the relevant chapter.

Chapter 1: Introduction

Chapter 1 begins by describing the challenge and its origins. The current need for effective tools to debug parallel programs is an outcome of the history of modern computing. Driven by advances in both theory and technology, computing grew from a niche mathematical curiosity to a powerful tool for the masses. Every year, new computers were cheaper, faster, and easier to program; they also required more power to operate. Eventually, this ever-growing power consumption became unsustainable, limiting improvements in computer performance. Computer engineers responded by switching to more power-efficient concurrent designs. Unfortunately, parallel systems have proven to be significantly harder to program than sequential systems. Despite considerable effort to develop new techniques to help write correct parallel programs, there remains a significant need for effective parallel debugging tools. This chapter discusses some of the proposed solutions to this challenge, and concludes with a statement of the goals that drove this work and motivated this thesis.

Chapter 2: The passert assertion language

Chapter 2 discusses the limitations of existing debugging tools which can be used for parallel programs. Sequential assertions are very effective for debugging sequential programs but ineffective for debugging parallel programs. Data-race and atomicity violation detectors are designed for use on parallel programs but suffer from high rates of false positive bug detections. More powerful tools such as theorem proving

---

1Throughout this thesis, the words concurrent and parallel are used interchangeably.
and model checking gain accuracy at the cost of complexity. They typically do not scale to large programs, and often require a Ph.D. to use correctly.

In response to this problem, I developed parallel assertions which are designed to be effective for debugging parallel programs. Parallel assertions form a simple yet powerful assertion language which allows programmers to express correctness conditions for parallel programs. I formally define the syntax of the parallel assertion language, and give several examples of its use.

Chapter 3: The Semantics of Parallel Assertions

Chapter 3 describes the semantics of the parallel assertion statement. Many modern processors implement weak memory models, which allow operations to occur out of order. These weak memory models improve the efficiency of the processor, but make it harder for the programmer to reason about the possible executions of their program. This creates a tension: many parallel verification tools assume a sequentially consistent model in which all operations must occur in program order, while current processors implement a different, weak memory model.

I therefore present two complementary parallel assertion semantics. The first describes the behaviour of parallel assertions on a sequentially consistent system. It is designed to make it easier to reason about what executions are possible for a given program. The second describes the behaviour of parallel assertions on a modern weak-memory system. It allows for a precise description of how parallel assertions interact with modern hardware, at the cost of an increase in mathematical complexity.

Chapter 4: Implementation

Chapter 4 describes PASSERT, the debugging tool I developed which uses parallel assertions to debug parallel programs. In particular, PASSERT allows programmers to
annotate their program with parallel assertions and then detect violations of these assertions as the program runs.

Running a debugging tool can significantly slow down the execution of the program under test. I implemented, and proved correct, a set of optimizations which dramatically improve the performance of PASSERT. Currently the performance overhead of PASSERT is competitive with similar debugging tools.

Chapter 5: Real World Applicability

The true test of an assertion language is its effectiveness at detecting and diagnosing real bugs in real programs. In Chapter 5, I evaluate the effectiveness of parallel assertions by applying them to the University of Michigan’s Collection of Concurrency Bugs — 17 real-world concurrency bugs from major open-source programs. In 14 cases, I was able to use parallel assertions to identify the cause of the bug. In 3 cases, I was unable to do so. Based on this experience, I propose extensions to the parallel assertion language to help cover these bugs.

Chapter 6: Conclusions

The final chapter of my thesis looks back at the challenges and successes of this work. It suggests future directions for improving both parallel assertions as a tool, and for the field of parallel debugging in general.

1.3 Background

1.3.1 The democratization of computing

For most of human history, “computers” were highly trained humans who computed the answers to mathematical problems. Attempts to mechanize this process date back to the ancient Greeks, but it was not until 1834 that Charles Babbage first designed
a fully programmable calculating machine, which he called the *analytical engine*. The architecture of Babbage’s design was strikingly modern: the analytical engine was “a general purpose digital computer consisting of a store, arithmetic unit, punched card input and output, and a card controlled sequencing mechanism that provided iteration and conditional branching” [83]. Lady Augusta Ada Lovelace became the first computer programmer when she wrote an algorithm to calculate Bernoulli numbers using an analytic engine [37]. Unfortunately, Babbage’s designs “turned out to be considerably beyond the technological capabilities of the era” [83]. Although both Babbage and the British government devoted considerable sums of money attempting to build his machines, none of these projects were ever successfully completed.

More than a century later, in 1937, Alan Turing laid the theoretical foundations for modern computer science in a paper with the daunting title “On Computable Numbers, with an Application to the Entscheidungsproblem” [96]. In contrast to Babbage’s description of the computer as a particular machine, Turing conceptualized it as any machine which could implement a specified set of operations. Any such machine could execute an algorithm written for any other computer. Alan Turing’s paper was powerful, elegant, and could only be understood by mathematicians with Ph.D.s.

The first general purpose electronic computer, called ENIAC (Electronic Numerical Integrator And Computer), was built in 1946, weighed over 30 tons, consumed almost 200 kilowatts of electrical power, and cost $486,804.22 ($5,774,671.37 in 2012 dollars). This powerful machine was capable of a previously unimaginable 5,000 additions per second. It was built by the University of Pennsylvania, and was used by the army to calculate ballistics tables [99].

In contrast to the ENIAC, which cost millions of dollars to calculate thousands of operations per second, today’s computers cost hundreds of dollars to calculate...
billions of operations per second. This has allowed a democratization of the computer, progressing from experimental machines in university laboratories, to mainframes at large corporations, to personal computers on office desks, to home computers and laptops. Today, people think nothing of carrying powerful computers around in their pockets — although they often refer to them as phones.

Moore’s law

Moore’s law describes this exponential decrease in price/performance of electronic hardware.\(^3\)

Moore’s original 1965 observation was that:

The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. . . . Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. \(^68\)

Despite the fact that the original formulation of Moore’s law was based on extrapolation from five data-points, it has accurately predicted computing advances for almost half a century. The timescale has increased slightly — the doubling period is now estimated at between eighteen and twenty-four months — but the rate of improvement has remained exponential \(^69\).

Structured programming

This growth in the capabilities of computer hardware has been matched by a similar explosion in both the complexity and the diversity of software. Microsoft Basic, for example, increased from 4000 lines of code in 1975 to half a million two decades later;\(^6\)

\(^3\)Every thesis on parallel programming requires at least one reference to Moore’s law.
Microsoft Word increased from 27,000 to two million lines of code over the course of a decade [87]. In July 2008, there were 800 apps on the iPhone app store; in September 2012, there were approximately 700,000 [26].

There is often a trade-off between programmer productivity and program efficiency. In other words, programmers typically have a choice between using high-level programming tools to quickly develop inefficient programs, or using low-level tools to slowly develop efficient programs. Until recently, Moore’s law reduced the sting of this trade-off (the reasons for the decline in Moore’s law improvements will be discussed in the next section). It was of little consequence if the inefficiencies introduced by new abstractions slowed program execution — Moore’s law meant that in just over a year, new hardware would allow the program to run twice as fast. This allowed programmers to trade (abundant) CPU time for (relatively scarce) human time.

High-level languages such as C provide simple but powerful abstractions which allow programmers to be more productive. A physical computer may implement complicated machine code, with control handled by unstructured goto statements, multiple concurrently active functional units, and memory separated into multiple disjoint segments. Languages such as C abstract this complicated machine and allow programmers to write programs for a sequential, structured language operating on a flat memory space. At every execution step, a well-formed sequential program has a single well defined program state, which makes it easier to reason about formally using techniques such as Hoare logic [46]. Sequential programs are deterministic and repeatable, which makes them easier to test and debug. Structured programs limit the set of program operations that could affect a control flow path, making control flow easier to reason about [30]. Modularity and object orientation do the same for data-flow.

\footnote{As I discuss in § 3.2 ill-formed C programs may have undefined behaviour.}
These languages have also proven effective at providing the performance that programmers require. The C machine model can be efficiently implemented on modern computers with low overhead. A positive feedback loop developed: as languages like C became more popular, computer architects delivered new CPU designs that were optimized for the C programming model, for example by deprecating features like segmented address spaces, which do not fit the flat memory model used in C/C++ [25]. This in turn improved the relative performance of C-like languages compared to other programming paradigms, and further boosted their popularity.

1.3.2 The power wall

If something cannot go on forever, it will stop. [94]

For almost half a century, computer scientists and engineers operated in a world full of free lunches. A programmer whose program was too slow, or too resource intensive, could simply wait. In eighteen months, following Moore’s law, computers would process twice as fast, have twice the memory, and use less power, all for the same cost. In 1995, Moore wrote:

By making things smaller, everything gets better simultaneously. There is little need for trade-offs. The speed of our products goes up, the power consumption goes down, system reliability, as we put more of the system on a chip, improves by leaps and bounds, but especially the cost of doing things electronically drops as a result of the technology. [69]

Moore left out one important fact in this rosy prediction: although Moore’s law scaling reduced the power consumption per transistor, it also enabled circuit designers to operate exponentially more complex chips at exponentially higher frequencies, which led to exponentially higher power consumption.

8
The power equation

The power consumption of a chip ($P$) depends on three things:

- $C$ The capacitance of the chip, i.e., its ability to store electric charge.
- $V$ The voltage at which the chip runs.
- $f$ The frequency at which the chip runs.

according to the relationship $P \propto CV^2f$.

For a given transistor technology, increasing the number of transistors causes a corresponding increase in the capacitance of the chip, and hence a corresponding increase in power consumption. Similarly, increasing the frequency at which the chip operates causes both a direct increase in power consumption, because power varies with frequency, and an indirect increase, because the maximum frequency of a transistor is a function of its operating voltage. Increasing the frequency of a chip also requires increasing its operating voltage, which causes a further increase in power consumption.
The net result is that in 1987, an Intel processor consumed one Watt; fifteen years later in 2003, a new Intel processor consumed 100 Watts. Computers were in danger of producing heat faster than it could be efficiently removed.

**Concurrent hardware to the rescue**

Consider a computer which has to complete two tasks as fast as possible. One approach is to perform each task as quickly as possible. If one task takes time $T$ to complete, then both tasks can be completed in time $2T$. A second approach is to perform the tasks more slowly, but to do multiple tasks in parallel. If each task in this approach takes $2T$ time to complete, but both tasks complete in parallel, then the total job will be finished in $2T$ units of time no matter which approach is employed. The power required by the two approaches, however, is quite different.

**Power consumption of fast processors** In order to run a processor as fast as possible, computer engineers have to make a number of power-hungry decisions.

They have to increase the voltage at which the chip operates. The speed at which a transistor switches is roughly proportional to the voltage at which it operates, so higher performing chips require higher voltages. The power consumed by a chip scales with the *square* of the voltage. A linear increase in frequency ($f' = \kappa f$) necessitates a corresponding linear increase in voltage ($V' = \kappa V$), and hence a *cubic* increase in total power consumption ($P' = \kappa^3 P$).

At some point, voltage scaling stops leading to increased transistor speed: there is a limit to how fast a transistor can switch. The speed at which a chip can run is capped by the length of the longest path in the chip, so computer architects break long computations down into pipelined stages, each of which can run quickly. In order to keep these pipelines operating effectively, computer engineers add compli-
cated structures such as branch predictors and reordering buffers. This dramatically increases the capacitance of the chip, and hence its power consumption.

Even with these structures, highly pipelined designs are (relatively) inefficient — Intel suggests that a 50% increase in frequency leads to a 30% increase in net performance [45]. A conservative estimate based on this assumption suggests that doubling the performance of a processor requires 2(1.5/1.3) = 2.3× increase in frequency.

Putting these elements together, doubling the performance of a single processor requires a 2.3× increase in frequency, which requires a corresponding 2.3× increase in voltage. Even ignoring the concomitant increase in chip complexity (and hence in chip capacitance), doubling the performance of a chip leads to a 12× increase in power consumption.

**Power consumption of parallel processors** Replacing a single core with two identical cores which run at the same frequency and voltage leads to a 2× increase in capacitance, and hence a 2× increase in power consumption.

**The implications for computer design** The two approaches calculate the same number of tasks in the same amount of time, and hence have the same throughput. The power consumption of the two approaches, however, is very different: the “single fast processor” approach uses 6× more power than the “two slow processors” approach. Most modern computers are designed with multiple, low-power processors to take advantage of this fact.

### 1.3.3 The concurrent programming challenge

The performance gains from the concurrent hardware described in the previous section will only materialize if programs can be effectively divided into multiple tasks.
which can execute in parallel. One approach is to take sequential programs and automatically partition them into parallel tasks. This already occurs to a limited level on modern CPUs, which have complex hardware structures to detect instructions which can execute in parallel [42]. Parallelizing compilers attempt to do the same for software [15]. While these approaches show some promise, they only work well at detecting certain common patterns. Taking full advantage of concurrent hardware still requires human ingenuity and input.

The multi-threaded programming model

A multi-core chip has two or more independent CPUs which can concurrently execute programs. One simple way to express a parallel program, therefore, is as a collection of sequential programs, each of which can execute on a different CPU. The simplest way for these programs to communicate is for them to share the same memory: when one program writes a value, another concurrent program can simply read it from the same memory location. Sequential programs that communicate through shared-memory in this way are called threads, and are directly supported in the C language by the Pthreads standard library.

In addition to communicating values, threads may need to synchronize their computation. Thread libraries provide synchronization primitives such as mutexes which ensure that only one thread can execute a computation at a time, barriers which cause a thread to wait until all other threads are ready, and signals which can be used to alert a thread that another thread wishes to communicate with it.

Why multi-threaded programming is hard

Superficially, sequential and multi-threaded programs appear almost identical. A parallel program is a collection of sequential programs, and it would appear that writing and debugging a parallel program should be no more difficult than writing
and debugging several sequential programs. In fact, there is a key difference. Sequen-
tial programs execute in a simple, deterministic manner which simplifies both
testing and debugging; in multi-threaded programs, *shared variables can be modified*
*by other threads*. This makes it difficult to reason about multi-threaded programs,
since program state can change values without the thread under analysis taking any
action. Making matters worse, this effect may occur non-deterministically, depending
on the order in which thread events are scheduled. This makes it difficult to test and
debug programs, since two runs of the same program with the same input may lead
to different results.

Correct programming has been described as the process of “establishing invariants,
perturbing them, and then re-establishing them. It is a game of stepping-stone from
one island of consistency to another” [21]. In parallel programming, the islands can
move underfoot.

It is possible to re-establish determinism in a parallel program through careful
use of synchronization. However, this is extremely difficult and even experts often
publish incorrect parallel algorithms. Concurrent programs are inherently harder to
write and debug than sequential programs.

**Compiler optimizations** The challenge of multi-threaded programming is exac-
terbated by the fact that common compiler optimizations can break multi-threaded
code [12]. For example, one common optimization is to replace a *switch* statement
with a jump table, as shown in Figure 1.2. On a sequential system, this code is
safe: the check that \((x < 2)\) guarantees that *table*[\(x\)] will be within bounds. On a
parallel system, however, some other thread can modify the value of \(x\) between the
check and the use, as shown on line 4 of the example. This would lead to *table*[\(x\)]
being an out of bounds dereference, and arbitrary code execution.

---

See, for example, the errata pages for Herlihy and Shavit’s standard text-book on multi-threaded
Original code:

```c
if (x < 2) {
    switch (x) {
        case 0:
            foo(); break;
        case 1:
            bar(); break;
    }
}
```

Compiles to:

```c
void (*table)(void) = {foo, bar};

if (x < 2) {
    //some other thread can set x = 3, causing wild code execution
    table[x]();
}
```

Figure 1.2: Compiler optimization causes unexpected results on multi-threaded code

1.4 How can computer science help?

The problems discussed above were tolerable when parallel programming was the domain only of elite experts such as computational physicists and operating system kernel designers. However, with the advent of multi-core processors in home computers, and now even in phones, it is no longer sufficient for parallel programming to be so hard that it takes a Ph.D. to get it right.

One approach to this problem is to make it easier to write correct parallel programs. Libraries such as OpenMP [76] or the Parallel Patterns Library [19] attempt to hide the difficulties of non-determinism, providing a simpler abstraction for programmer use. Strong type systems can guarantee race freedom in properly typed programs [39]. Flanagan and Qadeer similarly present a type system for guaranteeing atomicity [36].

Deterministic parallel programming removes the non-determinism from concurrent programs, albeit often at a performance cost [115, 57]. Functional programming attempts to make parallel programming easier by reducing the importance of mutable state [2]. Transactional memory allows programmers to separate their program into
composable atomic sections [43]. Message passing [41] and stream programming [95] bring order to chaos by enforcing structured communication patterns between threads. Automatic parallelization goes a step further, and extracts parallel code from sequential algorithms [15].

Despite the promise of these approaches, there remains a large body of legacy code. This code needs to be debugged. The goal of this thesis is to answer the question: “How can we help ordinary programmers debug parallel programs?”

1.5 Goals of this thesis

Like Calvin in the opening comic, computer architects have discovered the power of parallelism. Unfortunately, as computer scientists have learned, and as Calvin is about to discover, creating concurrency is easy; managing it effectively is hard. The goal of this dissertation is to solve this problem. In particular, is it possible to develop a debugging tool for multi-threaded programs which:

- Is simple to use?
- Is powerful enough to capture real bugs?
- Can be efficiently implemented on current computers?

This thesis is my answer to these questions.
Chapter 2

The PASSERT assertion language

2.1 Introduction

A programmer writing a piece of code has a set of assumptions about how the program will execute. The purpose of an assertion language is to allow the programmer to express these assumptions formally, as assertions embedded within their program. This chapter describes the PASSERT assertion language, a minimal extension to the C/C++ assertion languages which allows programmers to easily express correctness criteria for parallel programs.

I begin with an analysis of current industry standard for assertions in imperative languages such as C/C++, and in § 2.3 identify three key factors that have made them a widely used debugging tool for sequential programs. I then show, through
a series of examples in §2.4 why sequential assertions are not suitable for parallel programming. Guided by both the strengths and weaknesses of sequential assertions, I introduce a new assertion language which provides a simple, understandable set of predicates that allows testing of complex interleaved programs. To distinguish between the two types of assertions, I name the existing form *sequential assertions* and my new form *parallel assertions*.

The basic form of both sequential and parallel assertions is language agnostic. Sequential assertions can exist in any language that supports an `if` statement. Similarly, parallel assertions can, in principle, be specified in any multi-threaded language with syntactic scopes. This chapter simplifies the presentation of these assertions by focusing on C/C++ language examples and implementation.

### 2.2 What are sequential assertions?

A sequential assertion is a Boolean predicate about the state of a program at a particular point in time. If this predicate is violated, an exception is raised. If $\phi$ is a Boolean predicate on the state of the current program, the statement `assert($\phi$)` is effectively equivalent to the statement:

```c
if (!$\phi$) {
    abort();
}
```

In C/C++, for example, an assertion is syntactic sugar for a simple conditional. Figure 2.1 shows an example of how assertions are implemented in a standard open-source library.
2.2.1 Example of sequential assertion use

A simple C/C++ sequential assertion is shown in Figure 2.2. In this example, attempting to access a buffer using a NULL pointer represents an error. The assertion on line 1 will raise an exception if this happens during program execution.

2.3 Sequential assertions work for sequential programs

Sequential assertions are a widely used debugging technique. A recent software engineering study, for example, found that 5.1% of the lines of code in the studied open-source programs consisted of assertion statements [20]. Assertions are commonly used at Microsoft [27], and are encouraged in practically every coding standard and programming textbook. The question is: what makes assertions so prevalent? My analysis suggests three properties that have lead to the widespread use of sequential assertions.

2.3.1 Sequential assertions are easy to add

The easier it is for a programmer to add an assertion, the more likely they are to do so. Sequential assertions have three key features that make them easy to add. First of all, sequential assertions use a simple, familiar syntax. A programmer who is capable of writing an if statement in C/C++ is capable of writing an assertion statement in C/C++.
Secondly, sequential assertions enable local reasoning. A programmer adding an assertion about the value of a global variable does not need to care about the implementation details of the code that might affect that variable; they merely add a local check that the variable has a legal value when their code uses it. A well designed assertion language should maintain these properties.

Finally, the semantics of sequential assertions are simple, clear, and close to the semantics of the program they protect. This makes it make it easier for programmers to reason about the meaning of sequential assertions they use.

### 2.3.2 Sequential assertions are expressive

An assertion language is only useful if it enables programmers to express relevant correctness criteria, and hence to find bugs, on real programs. Kudrjavets et al. demonstrate a direct correlation between the use of sequential assertions and a decrease in bugs in real-world sequential programs [53].

Ideally, a language should be expressive enough to catch all bugs, and precise enough that any bugs detected by an assertion should reflect real bugs in a real program execution. A language which is theoretically more expressive, but which is too complex to use correctly, is of little value. Although many more powerful formalisms such as Linear Temporal Logic (LTL) [80] exist, the side-effect free first order logic used in sequential assertions has proven very effective.

### 2.3.3 Sequential assertions minimally affect code execution

Assertion checking can add delay to a program execution. If this delay is too great, programmers will not use assertions, no matter how theoretically useful they are. One key aspect of sequential assertions is that they can be used in debug builds, and then deactivated for release builds. A C/C++ release build with assertions deactivated has
the exact same performance as a C/C++ program without assertions. This means that end users of the program pay zero performance penalty for the use of assertions.

Ideally, it should be possible to remove assertions from production builds without affecting the correctness of the program, allowing programmers to test their program using a slow debug build, and then ship a faster production build. In practice, the use of assertions in this manner is potentially problematic because the released program can be different from the tested program. For example, the use of assertions might cause the compiler to use a different memory layout in the debug version, masking a buffer-overflow bug. A well-designed assertion language which only modifies the execution of the program under test in limited ways will minimize this effect.

2.4 Sequential assertions do not work for parallel programs

A recent paper called sequential assertions “one of the most useful automated techniques available for detecting faults and providing information about their locations” [23]. Unfortunately, while sequential assertions are highly effective at debugging sequential programs, they are surprisingly ineffective for debugging parallel programs. As I will show in the following section, many simple, common concurrency bugs are difficult or impossible to detect using sequential assertions.

In particular, sequential assertions are ineffective at debugging parallel programs for two reasons. They are not evaluated at the right time, and they cannot check the right properties.

2.4.1 Sequential assertions are not evaluated at the right times

Our intellectual powers are rather geared to master static relations and our powers to visualize processes evolving in time are relatively poorly de-
veloped. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible. [30]

A programmer writing a section of code has a set of assumptions about the environment in which that code operates. These can be as simple as that a pointer remains non-null, or more complicated, such as the requirement that a data-structure remain sorted during a search routine.

In a sequential program, the programmer can check this assumption by simply adding a sequential assertion checking the required property at the beginning of the relevant code block, and after every statement in the block that might invalidate the property. The combination of these assertion statements is sufficient to guarantee that any violation of the programmer’s assumptions will be detected. Figure 2.2 shows how a programmer might check for a null pointer error in a sequential program.

In a parallel program, however, this procedure is not sufficient, since a second thread could do an action which invalidates the assertion condition at any time. This reveals a fundamental problem with using sequential assertions in parallel programs: they are evaluated at one point in time, but the property they guarantee is used at a different point in time.

How to read the following examples: Each of the following examples shows an execution of two threads. Program events are represented as a sequence of C/C++ statements. For the purposes of these examples, each C/C++ statement executes as an atomic event. The columns represent threads — all events in the same column happened on the same thread. The rows represent an ordering of events — events in later rows happen after events in earlier rows. Figure 2.3 shows how an example
Is Equivalent To:

A: \texttt{stmt1}
B: \texttt{stmt2}
B: \texttt{stmt3}
A: \texttt{stmt4}

Figure 2.3: Translating a multi-column code layout into a sequentially consistent interleaved trace

in this layout format can be translated into an interleaved trace, i.e., placed into a single global order.

**Interference Problem**

Consider a programmer who wishes to ensure that a shared pointer is non-null before using it. Since assertion statements are not atomic with the code they protect, it is possible for another thread to modify the variable between the check and its use, as shown in Figure 2.4.

```
Thread A
assert(buffer != NULL);
copy_to_buffer(str, buffer);
```

```
Thread B
buffer = NULL;
```

ERROR: null buffer undetected

Figure 2.4: Interference Problem

One naïve attempt to solve this problem would be to put a sequential assertion before and after every statement in the target code-block. Figure 2.5 shows why such an approach is insufficient to reliably detect concurrency bugs. Since sequential assertions are not atomic with the code they aim to protect, it is possible
for a property to be violated between the check and its use, and then restored after
the error has occurred, but before the second check happens.

Thread A
assert(buffer != NULL);
copy_to_buffer(str,buffer);
assert(buffer != NULL);

Thread B
buffer = NULL;
buffer = &b;

ERROR: null buffer undetected

Figure 2.5: Multiple assertions cannot reliably detect the interference problem

2.4.2 Sequential assertions cannot check the right properties

Sequential assertions are predicates over the memory state of a system. However,
the state of a multi-threaded system includes both the current memory state and the
actions that other threads are taking. Parallel code makes assumptions about what
events are possible while the code is executing. These assumptions might be as simple
as that a statement executes atomically (Figure 2.6), or as complex as the ordering
requirements in a lock-free data structure (Figure 2.8). Figure 2.7 shows how an error
can occur even when the other thread does not modify shared state. The following
sections explain these examples in more detail, and show that there is no way to
specify properties involving such events with a standard sequential assertion.

Statement Atomicity Problem

A programmer writing parallel code has an expectation that certain statements will
execute atomically. However, even a simple increment instruction may actually be
executed as a series of loads and stores, with a potential for interference and incorrect
execution, as shown in Figure 2.6. This example is particularly difficult to debug
because from the local view of each thread, no obvious error occurred. Each thread
sees a memory state that started at \( x = 0 \), is incremented once, and ends with \( x = 1 \), which is, from that thread’s perspective, correct. It is only from the global perspective where the memory state started at \( x = 0 \), is incremented twice, and ends with \( x = 1 \), that the error is apparent.

Sequential assertions offer no mechanism for detecting the conflicting accesses that caused this bug.

Initially \( x = 0 \);

Thread A
\begin{align*}
\text{x++;} \\
\text{load r1,x} \\
\text{inc r1} \\
\text{store r1,x}
\end{align*}

Thread B
\begin{align*}
\text{x++;} \\
\text{load r2,x} \\
\text{inc r2} \\
\text{store r2,x}
\end{align*}

COMPILES TO:

\begin{align*}
\text{load r1,x} \\
\text{inc r1} \\
\text{store r1,x}
\end{align*}

\begin{align*}
\text{load r2,x} \\
\text{inc r2} \\
\text{store r2,x}
\end{align*}

ERROR: \( x = 1 \) (expected \( x = 2 \))

Figure 2.6: Statement Atomicity Problem

**Thread Safety Problem**

Programmers may add locks in an attempt to make accesses atomic. However, locks only work if every thread in the program always follows the correct locking discipline. In practise, programmers often forget to use locks, use the wrong locks, or use locks at an incorrect granularity, and hence fail to ensure the needed atomic sections. These bugs can be extremely difficult to detect, especially since locking violations can even occur in cases where the offending thread only reads the variable, as shown in Figure 2.7.

In this example, Thread B violates the lock and reads an inconsistent state from the buffer, but this is not reflected by any direct change to global state by Thread B. This makes it difficult for any assertion running on Thread A to detect the error, and will likely lead to an unexpected and difficult to diagnose failure later in the program.
Thread A
lock(buf.lock);
buf.val = foo;
buf.size = foosize;
unlock(buf.lock);

Thread B
//Cheating thread
//didn’t acquire lock
tempSize = buf.size;
tempVal = buff.val;

ERROR: Thread B read inconsistent state

Figure 2.7: Thread Safety Problem

ABA Problem

Locks are a fundamental bottleneck for parallel programs because they force the conservative serialization of potentially parallel operations. Instead, modern non-blocking algorithms observe an initial state, speculatively prepare an update, and then only apply that update if the initial state still holds, using atomic instructions like compareAndSet. The programmer’s intuition is that if an object has the same value at the end of the computation as at the beginning, no conflicting updates have occurred, and it is safe to commit the changes. Unfortunately, if the combination of delete and add operations restores the pointer to its original value, compareAndSet cannot distinguish between objects that have remained unmodified, and ones that have been modified twice, leading to a subtle bug called the ABA Problem.

For example, consider a non-blocking algorithm that maintains a sorted list (Figure 2.8). Thread A wants to add the value 4 to this list. First it finds the insertion point, which is the node with value 1, and records that the next node is the node with value 5 (highlighted with a heavy border). Ideally, it would then use compareAndSet to atomically insert a new node with value 4, maintaining the sorted list property.

Unfortunately, Thread B can delete the node with value 5, and then legally reinsert it with the value 3. Thread A still sees that the insertion point has a next pointer
Figure 2.8: The ABA Problem

which points to the node with the heavy border, and so makes the insertion, violating the sorted list property.

An assertion to catch this error needs to check that the `compareAndSet` succeeded only if the list was in the correct unmodified state at the precise point where the `compareAndSet` occurred. Since sequential assertions are not atomic with respect to the code they protect, they can miss this bug.

### 2.5 The Parallel Assertion Statement

In §2.3, I described how the simple but powerful design of sequential assertions has made them a successful tool for debugging sequential programs. In §2.4, I showed how they are ineffective for debugging parallel programs. This section presents my solution to this dilemma, which I call *parallel assertions*. My goal is that parallel assertions should be as easy to add, as expressive, and as low impact for parallel programs as sequential assertions are for sequential programs.

I accomplish this goal by minimally extending sequential assertions in a simple, intuitive manner. To solve the problem that sequential assertions are not checked at the correct time (i.e., are not atomic with the code they protect) (§2.4.1), I provide
thru {
    stmt;
    ...
    stmt;
} passert (passert_expr)

Figure 2.9: Parallel assertion scope and condition

da simple mechanism for a programmer to specify a scope during which the assertion must hold (§ 2.5.1). Similarly, I solve the problem that sequential assertions are not expressive enough to capture common parallel correctness criteria (§ 2.4.2) by adding new operators that allow programmers to express common parallel correctness criteria using a simple, intuitive, and familiar syntax (§ 2.5.2).

My experience annotating parallel programs suggests that the extensions I propose can cover a large range of properties of interest for parallel programs. More powerful formulations, such as temporal logic, are possible but are unfamiliar to programmers and provide expressiveness at the cost of complexity.

2.5.1 When they check: assertion scope

A parallel assertion defines a scope throughout which the assertion must hold. The parallel assertion scope solves the first problem with sequential assertions — they are checked at the wrong time — by unifying the assertion and the code it protects. In particular, the assertion scope is a block of code which delineates the time during which the assertion condition must hold. It begins with the keyword thru and ends with passert(passert_expr), as shown in Figure 2.9.

Parallel assertions can nest within each other, as shown in Figure 2.10. Although parallel assertion scopes are specified statically as syntactic code blocks, their evaluation at execution time is dynamic. Multiple threads can have active parallel assertion scopes at the same time, as shown in Figure 2.11. Intuitively, the lifetime of a parallel assertion scope is similar to that of an automatic (stack) variable which is defined
Figure 2.10: Nested parallel assertion scopes

in the first instruction of the scope, and potentially used by the last instruction in
the scope. Even if other functions are called, hiding a stack variable, it is alive (in
C++, its destructor will not be called) until the scope exits. Similarly, a parallel
assertion scope should remain live until its scope exits. If the program exits a syn-
tactic scope early, for example by executing a return statement, then the associated
parallel assertion scope should end as well.

Note that the condition must hold at all times, not merely during the times when
statements within the assertion scope are executing. The bug in Figure 2.8, for
example, occurs because of a delete and an add that occur on Thread B when no
actions were being executed in Thread A. In order to catch such bugs, I require
that even if the thread that the assertion is associated with is swapped out by the
scheduler, the assertion must continue to hold until the final statement within the
assertion scope.
2.5.2 What they check: assertion condition

I increase the expressiveness of parallel assertions by extending the assertion condition with a set of new operators which allow programmers to express conditions about read and write actions of both local and remote threads. These variables alleviate the limited expressiveness of sequential assertions by providing the programmer with the ability to express conditions about the actions of other threads.

Just as in sequential assertions, a parallel assertion condition can contain any standard side-effect-free Boolean expression. In addition, a parallel assertion condition can contain any of the operators specified in Figure 2.12 which allow programmers to capture when, by which threads, and in which order variables are accessed. A number of possible assertion conditions are given in Figure 2.13.

2.5.3 Formal Grammar

A formal grammar for parallel assertions in C/C++, based on the C++ grammar defined in [49, §A.4-5], is given in Figure 2.15. Unless otherwise specified, all operations have the standard C++ meanings, with the standard rules for operator precedence and type correctness.
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Read</td>
<td>True when the asserting thread is reading the location $\ell$</td>
<td>LR($\ell$)</td>
</tr>
<tr>
<td>Local Write</td>
<td>True when the asserting thread is writing the location $\ell$</td>
<td>LW($\ell$)</td>
</tr>
<tr>
<td>Remote Read</td>
<td>True when a thread other than the asserting thread is reading the location $\ell$</td>
<td>RR($\ell$)</td>
</tr>
<tr>
<td>Remote Write</td>
<td>True when a thread other than the asserting thread is writing the location $\ell$</td>
<td>RW($\ell$)</td>
</tr>
<tr>
<td>Has Occurred</td>
<td>True iff expr has ever been true while the assertion was active. Allows programmers to express happens before relationships. (See Figure 2.13)</td>
<td>HO(expr)</td>
</tr>
</tbody>
</table>

Figure 2.12: Parallel Assertion Operators

<table>
<thead>
<tr>
<th>Assertion Condition</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 passert(x == 5)</td>
<td></td>
</tr>
<tr>
<td>2 passert(x[y] &gt; *z)</td>
<td></td>
</tr>
<tr>
<td>3 passert(isprime(x))</td>
<td></td>
</tr>
<tr>
<td>4 passert(!RR(x))</td>
<td>no other thread reads x</td>
</tr>
<tr>
<td>5 passert(!RW(x-&gt;y))</td>
<td>no other thread writes x-&gt;y</td>
</tr>
<tr>
<td>6 passert(!RR(x)</td>
<td></td>
</tr>
<tr>
<td>7 passert(x.lock != 0</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.13: Example assertion conditions

**thru-statement**

The grammar for the *thru-statement* is based on that of the *do-while* iteration statement. C++ statements can contain optional special attributes — there are currently no special attributes associated with the *thru-statement*. Note that a thru-statement can recursively contain any statement type, including other nested thru-statements.

**Operators, lvalues, and rvalues**

The underlying C++ grammar is fairly complex, with many different types and subtypes of values and operators. Following the example of the C++ standard, I abstract two types of operators that can be used in parallel assertion expressions. I use *unary-op* to refer to unary C++ operations such as logical negation (!), and *infix-op* to refer to binary operations such as addition (+).
Similarly, I abstract two types of values. The first type are \textit{lvalues}, so called because they can occur on the \textit{left} of an assignment statement. \textit{Lvalues} are references to a memory location in which a value can be stored. The second type are \textit{rvalues}, which can only be used on the \textit{right} of an assignment state. \textit{Rvalues} reflect an expression that yields a concrete value that can be stored into, or read from, a memory location. I use $\ell \in \mathbb{L}$ to refer to \textit{lvalues} and $v \in \mathbb{V}$ to refer to \textit{rvalues}.

Formally, the C++ standard \cite[§3.10]{cite} defines that the non-terminal \textit{lvalue} represents an expression that “designates \ldots {} an object”, and an \textit{rvalue} is “a value that is not associated with an object”. Note that an \textit{lvalue} can refer to any addressable data structure, ranging from a single byte to a complicated C++ object, as shown in Figure 2.14.

\textbf{Assertion Expression}

$AExpr$ is the set of all side-effect-free C++ expressions (defined in \cite[§A.4]{cite}) augmented with a number of operators (described below). Note that therefore $(\mathbb{L} \cup \mathbb{V}) \subset AExpr$.

The access operators LR, LW, RR, and RW in Figure 2.15 take a single \textit{lvalue} $\ell$ (which takes the form of a \textit{cast-expression} in \cite[§5.4]{cite}) as a parameter, and check for the occurrence of a memory access to the respective object, as described in Figure 2.12. Intuitively, an operator such as RR($x$) applies from the byte $\&x$ to the byte $\&x$+\texttt{sizeof}(x).

The operator HO takes an assertion as a parameter and returns a Boolean indicating whether this assertion evaluated to true at some point in the respective scope.
class T {
    int a;
    int b;
    Foo x;
};

T t;
T* p = &t;

thru {
    ...
} passert(!RW (t)); //detects all remote writes to the object t
    //including to a,b, and x

thru {
    ...
} passert(!RW (*p)); //detects all remote writes to the object t;
    //including to a,b, and x

thru {
    ...
} passert(!RW (p)); //detects all remote writes to the pointer p
    //but not to to a,b, and x

thru {
    ...
} passert(!RW (p->x)); //detects all remote writes to the object x
    //but not to to a,b

Figure 2.14: Use of access operators on complex data-structures
2.6 Parallel assertions work for parallel programs

This section shows that for each bug in § 2.4 that could not be expressed using sequential assertions, there is a simple parallel assertion that captures the bug. Chapter 5 expands on this analysis, and shows that parallel assertions are applicable to a range of bugs in real open-source programs.

2.6.1 Interference

Consider the thread interference bug in Figure 2.4 which could not be detected using sequential assertions because they are not atomic with the statements they protect. The parallel assertion in Figure 2.16 would detect this bug.

```
Thread A
thru {
    copy_to_buffer(str,buffer);
} assert(buffer != NULL);
```

Thread B

```
buffer = NULL;
```

Figure 2.16: Assertion to detect interference bug
2.6.2 Statement Atomicity

The atomicity violation in Figure 2.6 could be detected using the parallel assertion in Figure 2.17. Notice that a hard to express global property (that no increment be lost) is converted into a set of easily expressible local properties.

Initially x = 0;

Initially x = 0;

Thread A

thru {
    x++; x++;
}

Thread B

x++;

Figure 2.17: Assertion to detect statement atomicity bug

2.6.3 Thread Safety

The locking violation in Figure 2.7 can be detected using a simple check that no other thread read or wrote the protected variables, as shown in Figure 2.18. Since this assertion checks reads as well as writes, it allows the programmer writing thread A to detect a violation even in the case where no shared program state was modified.

Thread A

lock(buf.lock);

through {
    buf.val = foo;
    buf.size = foosize;
}
passert(!RR(buf)&&!RW(buf));

Thread B

lock(buf.lock);

through {
    buf.val = foo;
    buf.size = foosize;
}
passert(!RR(buf)&&!RW(buf));

Figure 2.18: Assertion to detect thread safety bug
2.6.4 ABA Problem

To see the value of \texttt{HasOccurred} (HO), consider the ABA problem described in Figure 2.8. In this case, the program is correct if either no other thread modified the pointer, or if that modification was detected and the asserting thread aborted the write. This condition can be expressed as shown in Figure 2.19. Notice that this example also shows the need for the LW operator in addition to the RW operator, since the error occurs as a result of the ordering between local and remote events.

\begin{verbatim}
Thread A
cur = getInsertionPoint(4);
thru {
  tempnext = cur->next;
  tempnode = makeNode(4,tempnext);
  compareAndSet(cur->next,tempnext,tempnode);
} passert(!LW(cur->next)||HO(RW(cur->next)));

Thread B
delete(5);
add(3);
\end{verbatim}

Figure 2.19: Assertion to detect the ABA bug

2.7 Related Work

2.7.1 Languages for correct by construction parallel code

There has been a wide variety of work on tools to help programmers write correct parallel code. Some of this work focuses on new languages and runtime systems to ensure that parallel programs are correct by construction. Work along this line includes transactional memory systems [64], concurrent functional programming languages [2], stream programming [95], and automatic parallelization techniques [15]. While this work is valuable, the widespread adoption of threaded programming models and the large amounts of legacy code in imperative threaded languages suggest the need for tools to verify multi-threaded C/C++ programs.
2.7.2 Specification languages for debugging parallel code

There are a number of tools that have been developed to help programmers debug parallel programs. Ideally, these tools should provide maximal debugging power with minimal programmer effort. In practise, there is a trade-off between the completeness of a program specification, and the ease with which that specification can be developed.

At one end of the scale are verification tools that use implicit correctness criteria, and simply check for conditions that often represent concurrency bugs. Some of these tools, such as Goldilocks [31], FastTrack [35], and SigRace [71], check for race conditions. Others, such as the lockset algorithm, check to ensure that locks are used in a standard, consistent manner [80]. O’Callahan and Choi present a tool which minimizes false positives by combining race detection and lockset based detection [74]. Vlachos et al. propose a tool that uses taint analysis to check parallel programs for potentially dangerous uses of unsanitized input [98]. Lucia et al. weaken the data-race freedom condition and check for conflict freedom of synchronization-free regions, which they argue can be efficiently guaranteed at runtime [60].

The advantage of an implicit specification is that the programmer does not have to decide exactly what properties should be checked; the disadvantage is that the programmer may not know exactly what properties are being checked. Even within the verification community, there is a lack of consensus about the meaning of “data-race freedom” and “atomicity”: both terms have multiple inconsistent definitions. For example, “atomicity” can be used to refer to transactions with or without indivisibility guarantees. The term “data-race” is similarly ill-defined [72].

Even precisely specified implicit properties do not utilize the programmer’s knowledge about their code, which means that implicitly specified properties may not correspond to real bugs. Not all bugs are data races or atomicity violations: a program can be race free, but still calculate incorrect values or corrupt data. Not all data-
races are bugs: the canneal benchmark in the PARSEC suite, for example, deliberately allows data races because its probabilistic algorithm is capable of recovering from the errors they introduce [9]. Research suggests that only 2–10% of data races are harmful [103]. In general, tools with implicit specifications do well at catching standard bugs in standard programs. They do less well on programs that use non-standard techniques and tricks.

At the other extreme, approaches such as the Owicki-Gries method require the programmer to specify Hoare triples and invariants for all operations within their program [79]. Elmas, Tasiran and Qadeer require the programmer to create an abstract model of their program, and then check whether the program undergoing testing is a correct refinement of the model [32]. Theorem provers, such as Havoc, require the programmer to provide function contracts and loop invariants [6]. Havoc attempts to limit the need for programmer annotation by attempting to infer and then prove possible additional contracts from a partial specification. However, theorem proving tools still require the programmer to provide a relatively complete set of annotations. These are powerful tools in the hands of verification engineers. However, they require a considerable amount of skill and effort to use effectively, and it is unclear how useful they are to ordinary programmers.

In the middle are tools which provide programmers with a set of primitives that allow them to express common correctness criteria. Delgado et al. provide a useful taxonomy of such software monitors [28]. Kovacs et al. present a framework for writing full temporal assertions on message passing programs [52]. Although these temporal assertions are powerful, they are also complex. They require the programmer to write a Java class for each temporal property they wish to assert. Since a temporal logic property may depend on events that will happen in the future, their semantics introduces the concept of a partial tree. A temporal assertion which de-
pends on future events will have the value “unknown” (∗) until these events occur, limiting the use of temporal assertions for parallel runtime validation.

Vechev et al. [97] uses a set of Java primitives and the QVM virtual machine [3] to allow programmers to assert properties about object ownership in multi-threaded programs. If a thread knows that no other thread has a reference to an object, then it can know that no conflicting accesses will be made to that object.

**JASS** extends Java by allowing programmers to express parallel correctness conditions as conditions on objects [7]. Object method calls can be annotated with pre and post-conditions. Objects can also be annotated with class invariants, trace invariants, and refinement relations. Properties are checked when the object is stable, i.e., not being modified by any method call. These techniques are designed for systems like Java, where accesses occur through method calls and where data privatization and object synchronization is enforced by a runtime system. They are less applicable in languages without these features, like C.

### 2.8 Chapter Summary

As Calvin is discovering, it can be difficult for one concurrent object to control the effects that another concurrent object can have on it. In the physical world, Calvin’s mom can step in to try to restore order; in the world of software, parallel assertions serve a similar function.

Parallel assertions provide a powerful and intuitive tool that allows programmers to express their knowledge about their code as parallel correctness criteria, using familiar syntax and a minimal set of new predicates. Parallel assertions are efficient to write, easy to reason about, and allow programmers to detect many different types of errors that could previously only be identified with specialized tools. They are also
designed to be efficient to implement, since the correctness of a parallel assertion at a given time depends only on the instrumented state of the program at that time.
Chapter 3

The Semantics of Parallel Assertions

An assertion semantics should have several features. First and foremost, any bugs detected by an assertion should reflect real bugs in a real program execution. This means that the presence of assertions should not allow any behaviours that were not possible in the program without the assertions. Similarly, it is equally important that the presence of assertions not prevent any behaviours that were possible in the program without the assertions. It should be possible to remove assertions from production builds without affecting the correctness of the program. A programmer should ideally be able to use a slow debug build to test their program, and then ship a faster production build. Ideally, assertions should have semantics which are close to the semantics of the program they protect.

---

1According to the Oxford English Dictionary, semantics is a noun, usually treated as singular. The word “semantics” is usually treated as singular in this dissertation.
On the other hand, assertions need to be easy to use. Programmers require assertion semantics which are simple and clear in order to be able to effectively reason about the meaning of the assertions they use.

Unfortunately, on modern multiprocessor systems, there is a tension between the requirement that semantics be simple and clear, and the requirement that they be close to the semantics of the programs they protect. For performance reasons, modern multiprocessors implement weak memory models, which can lead to programs having surprising and even counter-intuitive semantics (§ 3.1). Even such simple questions as “what is the current state of the system?” or “did event A happen before event B?” may have no answer on a relaxed memory consistency system. Making matters worse, the behaviour of some parallel C/C++ programs can be undefined, raising significant debugging challenges (§ 3.2).

One solution to this problem is to enforce sequential consistency (SC), which requires that all operations appear to execute in a single total order [55]. Sequential consistency allows a much simpler semantics, at the cost of disallowing some possible system behaviours. This is the model assumed by many parallel debugging tools, and is guaranteed (for most programs) by the C/C++ standards. I present a sequentially consistent semantics for parallel assertions in § 3.3. The other solution is to provide a semantics which formally describes the precise re-orderings which are allowed. This approach is allows accurate modelling of real-world systems, at the cost of an increase in mathematical complexity. I present a weak-memory model semantics of parallel assertions in § 3.4.

Technically, SC is a weak memory ordering model in which all re-orderings are prohibited. The SC semantics can therefore be considered as an instantiation of the weak memory model semantics on a sequentially consistent system. However, a pure SC semantics is simpler than a weak memory model semantics, and hence makes it easier to reason about what executions are possible for a given program.
3.1 Memory Ordering Models

3.1.1 Why memory ordering models are necessary

For performance reasons, modern processors often reorder events, violating the naïve programmer expectation that events occur in program order. For example, in Figure 3.1 the processor reorders the execution of the two read instructions to hide the latency of the cache miss.

Sequential Consistency

On a single processor machine, it is relatively easy to ensure that the program appears to execute in order. On a multi-processor machine, ensuring this property can be difficult. Some machines ensure sequential consistency (SC), which guarantees that the operations of each individual thread are globally observed in a sequential order consistent with the program order [55]. In practise, ensuring sequential consistency requires expensive synchronization.

Weak memory models

Due to the cost of enforcing a total order on all events, most modern processors do not guarantee sequential consistency. Instead, processors have weak memory models which allow some re-orderings to be visible to executing code. Weak memory models pose a significant challenge for program debugging, because there is no longer a single,
Figure 3.2: Assertion with different evaluation on SC and weak memory model

consistent view of program execution and program state. For example, consider the assertion shown in Figure 3.2. On a sequentially consistent machine, this property trivially holds (part a). On a system with a weak memory model, however, the situation in part b can occur: the assertion holds on Thread A, but the corresponding assertion on Thread B is violated. Part c shows the explanation: the processor executing Thread B reordered the two reads.

3.1.2 The granularity of memory ordering models

A memory ordering model defines a (potentially partial) order on a set of events, where an event represents an atomic action in the system. What constitutes such an atomic action can vary from system to system, and does not always intuitively match programmer expectations. For example, on a Complex Instruction Set Computer (CISC) machine such the x86 architecture, incrementing a variable is a single
assembly instruction \texttt{inc [x]}. Internally, however, this single assembly instruction is
executed as a series of microcode instructions as shown in Figure 2.6. My experiments
show that running two parallel loops which concurrently call the same increment in-
struction 1,000,000 times seldom leads to two million increments, since the concurrent
increment instructions can destructively conflict. This means that even a seemingly
simple machine-code instruction is not necessarily atomic. The following sections are
therefore defined in terms of primitive operations such as reads and writes, which are
guaranteed to be atomic on modern computer architectures.

Even the simplest models, such as sequential consistency, can permit surprising
results if the memory model granularity does not match the programmer’s expecta-
tions. For example, whether the behaviour in Figure 2.6 can occur in a sequentially
consistent model depends on whether the model assumes that the statement, or the
microcode operation, is the fundamental atomic unit.

3.1.3 How memory ordering models are characterized

There are two basic formalisms that have been developed to describe memory ordering
models.

**Operational memory ordering models**

Operational models describe memory ordering constraints from the point of view of
an abstract machine that enforces them. A given execution trace is possible if and
only if there exists an legal implementation of the abstract machine which would
generate that execution trace. This type of model is particularly useful for hardware
engineers, because it allows them to determine whether a particular machine is a legal
implementation of the memory ordering model specification. A good example of an
operational memory ordering model is the Sarkar et al. operational model for the
Power architecture [85].
<table>
<thead>
<tr>
<th>Name</th>
<th>Notation</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>program order</td>
<td>(m_1^{po} \rightarrow m_2)</td>
<td>per-processor total order</td>
</tr>
<tr>
<td>dependencies</td>
<td>(m_1^{dp} \rightarrow m_2)</td>
<td>dependencies</td>
</tr>
<tr>
<td>po-loc</td>
<td>(m_1^{po-loc} \rightarrow m_2)</td>
<td>program order restricted to the same location</td>
</tr>
<tr>
<td>preserved program order</td>
<td>(m_1^{ppro} \rightarrow m_2)</td>
<td>pairs maintained in program order</td>
</tr>
<tr>
<td>read-from map</td>
<td>(w^\text{rf} \rightarrow r)</td>
<td>links a write to a read reading its value</td>
</tr>
<tr>
<td>external read-from map</td>
<td>(w^\text{rfe} \rightarrow r)</td>
<td>(\rightarrow) between events on different processors</td>
</tr>
<tr>
<td>internal read-from map</td>
<td>(w^\text{rf} \rightarrow r)</td>
<td>(\rightarrow) between events on the same processor</td>
</tr>
<tr>
<td>global read-from map</td>
<td>(w^\text{grf} \rightarrow r)</td>
<td>(\rightarrow) considered global</td>
</tr>
<tr>
<td>write serialization</td>
<td>(w_1^\text{ws} \rightarrow w_2)</td>
<td>total order on writes to the same location</td>
</tr>
<tr>
<td>from-read map</td>
<td>(r^\text{fr} \rightarrow w)</td>
<td>(r) reads from a write preceding (w) in (\rightarrow)</td>
</tr>
<tr>
<td>barriers</td>
<td>(m_1^{ab} \rightarrow m_2)</td>
<td>ordering induced by barriers</td>
</tr>
<tr>
<td>global happens-before</td>
<td>(m_1^{gbb} \rightarrow m_2)</td>
<td>union of global relations</td>
</tr>
<tr>
<td>communication</td>
<td>(m_1^{com} \rightarrow m_2)</td>
<td>((m_1, m_2) \in (\rightarrow \cup \rightarrow \cup \rightarrow))</td>
</tr>
<tr>
<td>observation</td>
<td>(m_1^{obs_i} \rightarrow m_2)</td>
<td>processor (i) observes (m_1) before (m_2) (added for this thesis).</td>
</tr>
</tbody>
</table>

Figure 3.3: Edge labelling for memory ordering model [1]

**Axiomatic memory ordering models**

Axiomatic models describe memory ordering constraints from the point of view of the program executing under the model. Under this model, a program execution can be represented as a directed graph whose *vertices* represent *program events*, and whose directed *edges* represent *dependencies* between these program events. Edges are tagged with labels representing the type of dependency: the notation used by Alglave et al. is shown in Figure 3.3 [1]. The number of labels in this figure is a testament to the complexity of modern memory models — some of the most important are illustrated in Example [1]. Different formalisms may use slightly different notation for these edges, but the basic concept remains the same. In this thesis, I introduce an edge \(\rightarrow\) to represent an dependency induced by one thread observing two events as occurring in a given order. For clarity, I sometimes subscript edges to represent
which processor induced that edge. For example, for two events occurring in program order on processor $i$, I sometimes write $m_1 \xrightarrow{po} m_2$.

A given memory model will require that certain dependencies be respected, but will allow other dependencies to be ignored. Enforced edges create a *happens before* relationship between the events they connect: for example, if read-from map edges are enforced by a given memory model, and $w \xrightarrow{rf} r$, then $w$ must strictly happen before $r$.

Since it is impossible for an event to happen before itself, a program execution which induces a cycle in the happens-before graph is infeasible in that memory model; a program execution which does not induce a cycle is feasible.

Memory models differ in which dependencies are required, and which are ignored. The sequentially consistent model, for example, requires that all events must occur in program order: it enforces the $\xrightarrow{po}$ relation, and hence has $\xrightarrow{po} \equiv \xrightarrow{ppo}$. The x86 memory model enforces program order between pairs of reads ($\xrightarrow{ppo}$), and between pairs of writes ($\xrightarrow{ppo}$). It enforces program order dependencies between accesses to the same location ($\xrightarrow{po-loc}$). It does not enforce the dependency between reads and writes to different locations, and hence does not enforce $\xrightarrow{po}$. These subtle distinctions between memory models lead to the explosion of edge label types in Figure 3.3. A good example of an axiomatic memory ordering model is the Haim et al. axiomatic model for the Power architecture [62].
Example 1. The left hand side of Figure 3.4 shows a potential execution of a small program which would violate a parallel assertion. I consider its behaviour under a sequentially consistent model, and under a weaker model which does not enforce ordering between pairs of writes to different locations.

The right hand side of Figure 3.4 shows an event ordering graph for this small program. It contains a set of read events \((R_{\text{tid}} \ell v)\) and write events \((W_{\text{tid}} \ell v)\) where tid identifies the process that caused the event, \(\ell\) is the affected location, and \(v\) is the value. These events are connected by a number of different types of edges:

\((\rightarrow_p)\) These represent preserved program order edges, which are enforced under both memory ordering models.

\(e.g. (R_2 x 0 \rightarrow_p W_2 z 1)\)

\((\rightarrow_{fr})\) These represent a Write After Read (WAR) hazard. These events must occur in this order because of a data dependency, and are enforced in both memory ordering models.

\(e.g. (R_2 x 0 \rightarrow_{fr} W_1 x 1)\)

\((\rightarrow_{rf})\) These represent a Read After Write (RAW) hazard. These events must occur in this order because of a data dependency, and are enforced in both memory ordering models.

\(e.g. (W_1 y 1 \rightarrow_{rf} R_2 y 1)\)

\((\rightarrow)\) This represents an edge which may or may not be enforced, depending on the strictness of the model.

A platform which guarantees sequential consistency enforces \((W_1 x 1 \rightarrow_p W_1 y 1)\). In this case, the dotted line in the graph represents a binding constraint. This induces a cycle in the graph, and hence this execution is impossible and the assertion is not violated on a sequentially consistent platform.
A weak memory platform which permits reordering of write events does not enforce \((W_1 x 1 \xrightarrow{po} W_1 y 1)\). In this case, the dotted line in the graph does not represent a binding constraint, and the event graph is acyclic. The assertion can theoretically be violated on a relaxed memory platform.

### 3.1.4 Fences

In some cases, the re-orderings permitted by weak memory models are benign — they improve performance, and do not affect correctness. In other cases, the correctness of a parallel algorithm depends certain events occurring in the correct order. Processors therefore provide *fence* instructions, which enable the programmer (or compiler) to enforce a global ordering between events.

Different platforms provide different types of fences, and their semantics depends on the specific architecture (c.f. [1, 22]). In general, I distinguish between non-cumulative and cumulative fences. Intuitively, non-cumulative fences prevent thread-local reordering of events across the fence event. Cumulative barriers also affect the order of events of other threads (e.g., by flushing store buffers or caches). One non-cumulative fence operations is *mfence* on Intel processors: it “guarantees that every load and store instruction that precedes in program order the *mfence* instruction is globally visible\(^2\) before any load or store instruction that follows the *mfence* instruction is globally visible” [25, pg. 4-23].

Formally, two memory events \(e_1, e_2 \in \mathcal{M}\) occurring before and after a *mfence* event in program order, respectively, will be observed in this order by all threads:

\[
\exists n. \left( e_1 \xrightarrow{po} \text{mfence} \xrightarrow{po} e_2 \right) \Rightarrow \forall n. e_1 \xrightarrow{obs} e_2
\]  

\(^2\)The concept of global visibility coincides with our notion of observability.
The \texttt{hwsync} instruction of the Power architecture provides a similar guarantee, but is also cumulative in the sense that it separates the memory events observed by the thread before and after executing \texttt{hwsync}.

The distinction comes when observing events on different threads. If Thread A observes an event on Thread B, and then issues a \texttt{mfence}, it is possible for Thread C to observe the \texttt{mfence}, and then the event on Thread B. On the other hand, if Thread A observes an event on Thread B, and then issues a \texttt{hwsync}, then Thread C must also observe the event on Thread B before it observes the \texttt{hwsync} event.

\textbf{Example 2.} As discussed in Example 1, the assertion in Fig. 3.4 holds on any platform that guarantees sequential consistency, but may fail on a platform that permits reordering of write events. A fence between the events $W_1x1$ and $W_1y1$ would induce a $\rightarrow a^{b}$ edge (indicated by a dashed arrow in the figure) which rules out the failing execution and restores sequential consistency for this program.

The side effect described in Example 2 is not always desirable. If the fence instruction is part of the instrumentation code required to log the write events, then this modification effectively eliminates an erroneous execution. Logging mechanisms requiring stronger guarantees (as provided by the atomic compare-and-swap instruction \texttt{lock cmpxchg} on Intel architectures, for instance, which is frequently used to implement locks) exacerbate the probe effect.

A complete formalization of fences exceeds the scope of this work; I refer the reader to [1] instead.

\section*{3.2 C/C++ semantics}

Since parallel assertions are an extension of the C/C++ languages, a natural starting place for defining parallel assertion semantics is to start with C/C++ semantics. Until recently, neither C nor C++ had a well defined semantics when executing multi-
threaded programs [13]. Implementations were free to interpret parallel programs as they wished. Unfortunately, many common optimizations that are legal for single-threaded programs have unexpected consequences when used in a multi-threaded context [12]. This lack of standardization made it difficult for programmers to write correct parallel programs. A program which used to run correctly might display unexpected bugs after a compiler upgrade.

The adoption of the C++11 standard [49], and subsequent C11 standard [50], provide programmers with a precisely defined semantics for multi-threaded programs. Although there are complicating details, at a high level the standard guarantees that a programmer who follows certain well defined rules for synchronization of potentially conflicting accesses can assume that their program will appear to execute in a sequentially consistent manner.

The semantics is defined based on the concept of a “happens before” relationship, with data races defined as follows:

§1.10.4 Two expression evaluations conflict if one of them modifies a memory location and the other one accesses or modifies the same memory location. [49]

§1.10.21 The execution of a program contains a data race if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. Any such data race results in undefined behaviour. [49] Emphasis added]

Although this new semantics is both formal and precise, it is insufficient for defining the semantics for a parallel assertion language because it deliberately leaves the semantics for erroneous programs undefined. According to the standard:

§1.3.24 Undefined behaviour may be expected when this International Standard omits any explicit definition of behaviour or when a program
uses an erroneous construct or erroneous data. Permissible undefined behaviour ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message). [49]

A single error anywhere in the program, such as calling a library function with an invalid parameter or de-referencing an invalid pointer, potentially causes the entire program semantics to become undefined, with no warning required from the implementation. This is particularly applicable to parallel programs, because in such programs a data race results in undefined behaviour.

In practise a compiler would still generate assembly code (albeit with a compiler-specific behaviour) for such a program. It is exactly in such corner cases that parallel assertions enabling the programmer to debug the flawed program are particularly valuable. I therefore base parallel assertion semantics on an observable execution of the program under test.

### 3.3 Sequentially Consistent Semantics

I begin by introducing a sequentially consistent parallel assertion semantics. Although sequential consistency is not guaranteed on most modern processors, it does offer several advantages for a debugging system.

Sequentially consistent executions are (relatively) easy to reason about. A sequentially consistent execution has a single, globally ordered trace of events. Since every thread sees the same events in the same order, every thread observes the same global system state at every point in time. Event traces respect program order, making it

---

3I refer to observable executions instead of observed executions because the program may be analyzed symbolically.
easier to associate a given event trace with the program statements that caused it. Although some bugs (e.g., Figure 3.4) only occur on weak memory models, many others, such as atomicity violations due to insufficient locking, occur regardless of the memory model. As discussed in §3.2, a properly written data-race free C/C++ program will display sequentially consistent semantics (although, as discussed in §3.1.2 these semantics might still permit surprising executions due to the non-atomicity of C/C++ statements).

Sequentially consistent executions can also be easier to evaluate. A tool attempting to analyze a parallel program using a weak memory model needs to consider every possible reordering allowed by the memory model; a tool using a sequentially consistent semantics, on the other hand, need only consider in-order program execution. This makes sequential consistency a popular choice for parallel debugging tools.

3.3.1 Observable Program Execution Timeline

A parallel program \( \mathcal{P} \) consists of a set of threads, each of which generates a set of events, such as reads and writes. A program execution is an ordered interleaving of events caused by threads. A timeline \( \mathcal{T} \) is a globally observable total ordering of these events for a particular execution of a parallel program — i.e., every thread observes the same timeline \( \mathcal{T} \). I ensure such a total order by placing several requirements on any implementation of parallel assertions. In many cases, these requirements will be guaranteed by the underlying hardware; in cases where they are not, a parallel assertion checking implementation must ensure they hold throughout the implementation. Since some bugs only manifest on systems with weak memory models, it is possible that an assertion checker that enforces these rules may preclude certain errors from being detected.

Requirement 1: Events appear to occur \emph{atomically}. For some actions, like writes to single words, this will be guaranteed by the underlying hardware; in cases where it
is not, a parallel assertion implementation must ensure atomicity at the software level. Depending on the implementation, this may require the use of some combination of locks, fences, and atomic operations.

**Requirement 2:** All threads must observe the same consistent ordering of events, i.e. there must be a single total order. If Thread 1 observes event A as occurring before event B in timeline \( T \), then every thread in the timeline must observe event A before event B in timeline \( T \). This property is inherently true on systems with sufficiently strict memory models (such as sequential consistency); on other systems, a parallel assertion implementation may need to add appropriate memory fences to ensure a consistent ordering of events across multiple threads.

**Requirement 3:** There can be no intra-thread instruction reordering across a thru scope boundary. Scope begin/end must have acquire/release semantics — i.e. have implicit memory fences at both the hardware and programming language level.

These requirements restrict intra-thread optimizations, and may change the timing of a program, potentially changing its output. This “probing effect” is inherent to a debugging system — the act of observing a system changes its potential output. This effect occurs even with standard sequential assertions in sequential programs. A sequential assertion must access the variables it references, forcing the compiler to avoid optimizations to those variables that cross the assertion boundary, potentially changing program behaviour. The existence of a sequential assertion can even cause a program to change its memory layout, potentially hiding the effects of a buffer overflow bug. One of the key properties of sequential assertions is that while the probe effect changes whether a bug appears on a particular execution run, any bug found by an assertion represents a real bug, representing a condition that could actually occur during program execution. Thus, they are sound. I add a similar requirement to my sequentially consistent parallel assertion semantics:
**Requirement 4:** Event ordering must respect program semantics. Let $P$ be a program, $P_A$ the same program with assertions, and $T$ a timeline for $P_A$. Then there must be some legal execution of $P$ in which every read and every write occur in the same order, on the same threads, to the same locations, and have the same values as the read and write events in $T$. This requirement is trivially true for programs where the C/C++ semantics are undefined, such as programs with invalid pointer usage or data-races. As I discussed before (§ 3.2), in practise such programs are compiled into code which does have a meaningful semantics. The more closely a parallel assertion checker follows this implementation dependent behaviour, the more useful it will be for debugging.

Requirements 1–3 may require a stricter memory model than required by the underlying program semantics. As I discussed in § 3.1.4 strong memory models can be enforced on systems with weaker models by adding appropriate fences and synchronization. Since a strict memory model is a valid implementation of a weak memory model, an implementation can achieve requirements 1–3 without violating requirement 4.

**Timeline Events**

A timeline consists of an ordered series of events that meet requirements 1–4. Events may be associated with the data items described below:

- **Location** $\ell$ is a tuple $(address, VariableType)$ where $address$ is a memory address and $VariableType$ is a type in the underlying implementation, such as `int` or `struct foo*`. A location can be thought of as a set of bytes, ranging from $[address, address+\text{sizeof}(VariableType)]$. Two locations are equal if they have the same address and type. Two locations intersect if any of their bytes overlap.
void f(list* node)
{
    if (!node)
        return;
    thru{
        process(node->data);
        f(node->next);
    }assert(!RW(*node));
}

Figure 3.5: Single assertion with multiple dynamic $\phi_{uid}$

- **Value** $v$ represents a constant value that can be read or written from a memory location.

- **Thread ID** $tid$ identifies the the thread causing the event.

- **Parallel Assertion Condition** $\phi_{uid}$ is associated with the assertion condition of a particular dynamically active parallel assertion. Since multiple parallel assertions can be active simultaneously, it is possible that a single syntactic assertion may be related to multiple distinct $\phi_{uid}$. In Figure 3.5, for example, every time the function $f$ is recursively called, a new $\phi_{uid}$ with a new binding for $*node$ is dynamically created, even though there is only a single syntactic assertion $assert(!RW(*node))$ in the program.

There are many possible types of events that can be generated during program execution. The following four types of events are sufficient to allow the evaluation of parallel assertions. Notice that all of these events atomically complete during a single time-step, and hence are observed only during that time-step.

- $(READ, (\ell, v), tid)$
  A read by thread $tid$ returned value $v$ from location $\ell$. 

55
• \(\text{WRITE},(\ell,v),tid\)

A write by thread \(tid\) stored value \(v\) into location \(\ell\).

• \(\text{BEGIN}\_\text{PASSERT},\phi_{uid},tid\)

A parallel assertion with condition id \(\phi_{uid}\) has been triggered on thread \(tid\). i.e. assertion \(\phi_{uid}\) has entered its scope.

• \(\text{END}\_\text{PASSERT},\phi_{uid},tid\)

The parallel assertion with condition id \(\phi_{uid}\) on thread \(tid\) has completed, i.e. assertion \(\phi_{uid}\) has exited its scope. Note that \(\text{BEGIN}\) and \(\text{END}\) must form pairs in the execution trace. A trace in which this is not true is invalid and represents a fault in the implementation of parallel assertion checker. If the \text{PASSERT} dynamic runtime checker detects such a fault, it throws an exception and halts.

All actions should be atomic per requirement 1. For accesses to native data types such as \texttt{int}, this is true by default on most modern hardware. However, extended types such as \texttt{long long} may be accessed non-atomically. A parallel assertion checking implementation may treat these accesses as atomic, and raise an error if a conflicting update occurs that violates this atomicity, since this likely violates the programmer’s assumptions. Larger data-structures are inherently non-atomic, and it is the responsibility of the programmer to add parallel assertions that represent the desired types of atomicity that should be enforced on these structures.

**Timestamps**

The combination of requirements 1–3 ensures that every program execution has a consistent total ordering of events. Given a timeline \(\mathcal{T}\), every event can be uniquely associated with a time-stamp \(\text{Time}(\text{event})\) which reflects its order in the timeline. Thus, \(\text{Time}(e_a) < \text{Time}(e_b)\) iff \(e_a \rightarrow e_b\) in \(\mathcal{T}\) by all threads.
3.3.2 Instrumented State

On the one hand, it is convenient and natural to describe an assertion as a Boolean predicate over the state of a program. On the other hand, as I discussed in Chapter 2, the correctness of a parallel program can depend on when, by which threads, and in which orders variables are accessed. I resolve this difficulty by defining an Instrumented State which includes both the memory state at a time $t$, as well as variables describing the events occurring at that time $t$. Although for clarity I describe augmented state as unbounded arrays of variables, any implementation that can simulate these arrays is valid. An implementation which can use static or dynamic analysis to determine that only some portion of the total state is necessary to evaluate an assertion needs to track only that portion of the state.

The instrumented state $S$ of an instrumented program at any time $t$, denoted $S_t$, consists of three kinds of variables: program variables, observation variables, and history variables. I write $S_t.v$ for the value of variable $v$ at time $t$.

**Program Variables**

These are standard program variables, which represent the un-augmented state of the program.

- $M[\ell]$

  The program memory, represented as a mapping ($\ell, v$) which maps locations $\ell$ to values $v$.

**Observation Variables**

These variables encode which events are occurring at time $t$. Encoding events as indicator variables allows the parallel assertion condition to be expressed as a simple Boolean predicate over the instrumented state, making it easier and more familiar
for programmers. In particular, observation variables are the instrumented state necessary to evaluate LR, LW, RR, and RW operators.

In the following, $\phi_k$ refers to the $k^{th}$ unique assertion in the program. Since assertion scopes are dynamic, a single syntactic assertion may be dynamically associated with several distinct $\phi_k$, as shown in Figure 3.5.

- $R[\ell, tid]$
  True if the event at time $t$ was a read by thread $tid$ from location $\ell$, false otherwise.

- $W[\ell, tid]$
  True if the event at time $t$ was a write by thread $tid$ to location $\ell$, false otherwise

- $LIVE[\phi_k]$
  True at all time $t$ between (BEGIN_PASSERT, $\phi_k$, tid) and its corresponding (END_PASSERT, $\phi_k$, tid), false otherwise.

**History Variables**

The “Has Occurred” (HO) operator, described in § 2.5, allows assertions to reference a limited amount of history. History variables record the information necessary to evaluate a given usage of the HO operator at any point in time. In particular, each history variable records whether its argument, a Boolean predicate $\Omega(S_t)$, has ever been true.

A parallel program may contain multiple independent parallel assertions $\phi_k$, each of which will need an independent set of history variables. Each of these history variables will be associated with a different Boolean predicate on the instrumented state $\Omega_j$. All history variables associated with $\phi_k$ are reset to false when $\phi_k$ begins.

- $H[\phi_k][\Omega_j]$: $\exists \tau : \tau \leq t \land \Omega_j(S_\tau) \land S_\tau.LIVE[\phi_k]$.
  True if $\Omega_j$ is true now, or has ever been true in the past, while $\phi_k$ was live.
T(Event e, State $S_i$)
atomic {
  $S_{t+1} = S_i$;
  foreach($\ell$, tid)
    $S_{t+1}.R[\ell, tid] = S_{t+1}.W[\ell, tid] = false$;
  switch(e)
    case (READ,$\ell$, v, tid):
      $S_{t+1}.R[\ell, tid] = true$;
    case (WRITE,$\ell$, v, tid):
      $S_{t+1}.W[\ell, tid] = true$;
      $S_{t+1}.M[\ell] = v$;
    case (BEGIN_PASSERT,$\phi_k$, tid):
      $S_{t+1}.LIVE[\phi_k] = true$;
    case (END_PASSERT,$\phi_k$, tid):
      $S_{t+1}.LIVE[\phi_k] = false$;
  foreach($\phi$, $\Omega$)
    $S_{t+1}.H[\phi][\Omega] = S_t.H[\phi][\Omega] \lor (S_{t+1}.LIVE[\phi] \land \Omega(S_{t+1}))$;
  return $S_{t+1}$;
}

Figure 3.6: Instrumented State Transition Function

### 3.3.3 Instrumented State Transition Function

At program initialization, all auxiliary variables are initialized to false:

$$\forall \ell, tid, \phi_k, \Omega_j : R[\ell, tid] = W[\ell, tid] = LIVE[\phi_k] = H[\phi_k][\Omega_j] = false$$

The instrumented state is updated atomically according to a transition function $T(\text{Event}_i, S_i) \rightarrow S_{t+1}$ (Figure 3.6). Since read/write operations are observed only during a single time-step, this function begins by clearing any variables that represent read/write operations from the previous time-step. It then updates the program, observation, and history variables to reflect the actions that occurred during the current time.
3.3.4 Parallel Assertion Scope

I can now formally define the semantics of the parallel assertion scope which was informally defined in § [2.5.1]

A parallel assertion is associated with a defined block of program code, which I refer to as the parallel assertion scope. A parallel assertion is active from the first event that executes within the scope until the last event that executes within the scope. Formally, the event `{BEGIN_PASSERT, φ, tid}` occurs immediately before the first statement on thread `tid` that is within the scope i.e. there is no event on any thread in the timeline between this event and the next event in this thread. Similarly, the event `{END_PASSERT, φ, tid}` occur immediately after the end of the last statement executed within the scope on thread `tid`. This requirement refers to the last statement executed, which may differ from the last statement in program order if code exits the scope using a break or return statement. No matter how the scope is exited, `END_PASSERT` should be the unique exit to the scope. This may require an implementation to add code before early return, break, or goto statements, much like current C++ implementations already to do handle object destruction on scope exit.

A parallel assertion scope is dynamic — scopes can contain function calls and loops. Intuitively, the lifetime of a parallel assertion scope is similar to that of an automatic (stack) variable which is defined in the first instruction of the scope, and potentially used by the last instruction in the scope.

A parallel assertion holds if and only if its condition `φ` is true for all times during which the assertion is active.

Every scope is associated with a thread, and variable accesses are defined relative to that thread. Multiple threads may have active scopes, which may refer to the same sections of program code. Parallel assertions can nest, both directly and through calls
to functions which themselves contain parallel assertion scopes. There can therefore be many simultaneously active parallel assertion scopes.

**Important note:** *Inter*-thread behaviour is unaffected by the existence of a thru scope. A parallel assertion checks, but does not enforce, ordering of events between program threads.

### 3.3.5 Parallel Assertion Condition

This section gives a precise definition of the parallel assertion condition which was informally defined in § 2.5.2

An assertion condition $\phi$ is a Boolean predicate over an *instrumented state*, $S_t$, of a program at time $t$. As such, it can be any side-effect-free Boolean formula over the memory state of the program at time $t$, $S_t.M$. In addition, I define the following Boolean predicates that use the instrumented state, and may be used as sub-formulas in $\phi$. $tid$ is the thread id of the thread currently executing the parallel assertion.

- **Local Write** [Keyword LW($\ell$)]
  
  $S_t.W(\ell,tid)$

  True if the asserting thread is writing the location $\ell$.

- **Remote Write** [Keyword RW($\ell$)]

  $\exists thrd : tid \neq thrd \land S_t.W(\ell,thrd)$

  True if a thread other than asserting thread is writing the location $\ell$.

- **Local Read** [Keyword LR($\ell$)]

  $S_t.R(\ell,tid)$

  True if the asserting thread is reading the location $\ell$.

- **Remote Read** [Keyword RR($\ell$)]

  $\exists thrd : tid \neq thrd \land S_t.R(\ell,thrd)$

  True if a thread other than asserting thread is reading the location $\ell$. 
• Has Occurred \[ \text{Has Occurred} \ [\text{Keyword } \text{HO}(\Omega)] \]

\[ \mathcal{S}_t.H[\phi][\Omega] \]

HasOccurred functions as a latch (a simple memory element) — it takes a Boolean predicate \( \Omega \), and returns true if and only if \( \Omega \) is true now, or has ever been true in the past while this dynamic instance of the assertion was active.

### 3.3.6 Checking Parallel Assertions

An instrumented state \( \mathcal{S}_t \) contains all information necessary to evaluate a parallel assertion at time \( t \). An assertion \( \phi \) holds if \( \forall \tau : \mathcal{S}_\tau.LIVE[\phi] \Rightarrow \phi(\mathcal{S}_\tau) \); it fails otherwise.

### 3.3.7 Evaluation Example

The example in Figure 2.18 uses a parallel assertion to check whether a buffer is accessed in a thread-safe manner. A possible execution timeline, and the associated instrumented state, are shown in Table 3.7.

The assertion begins at time \( t = 2 \), following the acquisition of the lock at \( t = 1 \) (a combined read/write event). Until time \( t = 4 \), no other thread reads the buffer, and so the assertion holds. At time \( t = 4 \), thread 2 reads the buffer, and the assertion is violated, and similarly for time \( t = 5 \).

Note that at time \( t = 6 \), the assertion condition is once again true, since no other thread is accessing the buffer. However, this state is irrelevant since the assertion would have already failed at time 4.
Figure 3.7: Instrumented State for an execution of Example 2.18

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td>oldSize</td>
<td>oldVal</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>1</td>
<td>(READ, buf.lock,0,tid_A)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(WRITE, buf.lock,1,tid_A)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(BEGIN_PASSERT,φ,tid_A)</td>
<td>1</td>
<td>oldSize</td>
<td>oldVal</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>3</td>
<td>(WRITE, buf.val,foo,tid_A)</td>
<td>1</td>
<td>oldSize</td>
<td>foo</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>4</td>
<td>(READ, buf.size,oldSize,tid_B)</td>
<td>1</td>
<td>oldSize</td>
<td>foo</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>5</td>
<td>(READ, buf.val,foo,tid_B)</td>
<td>1</td>
<td>oldSize</td>
<td>foo</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>6</td>
<td>(WRITE, buf.size,foosize,tid_A)</td>
<td>1</td>
<td>foosize</td>
<td>foo</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>7</td>
<td>(END_PASSERT,φ,tid_A)</td>
<td>1</td>
<td>foosize</td>
<td>foo</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>8</td>
<td>(WRITE, buf.lock,0,tid_A)</td>
<td>0</td>
<td>foosize</td>
<td>foo</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>
3.4 Relaxed Ordering Semantics

Although sequentially consistent systems are easier to analyze, in practise most programs run on systems with weaker memory models. This causes two related problem for a sequentially consistent debugging system.

The first is that a sequentially consistent system may miss bugs that occur in real executions (e.g. Figure 3.4). The second is that enforcing sequential consistency on a weaker memory model system requires the use of expensive synchronization that slows down program execution. Sequentially consistent static checkers are faster; sequentially consistent runtime checkers are slower.

This section therefore introduces a relaxed ordering semantics for parallel assertions that is compatible with the weak memory ordering models used in modern processors. This relaxed ordering semantics enables two key optimizations which, as I demonstrate in Chapter 4, are crucial to making run-time checking of parallel assertions feasible.

The basic design of the relaxed ordering semantics is deliberately similar to the sequentially consistent semantics presented in § 3.3. The notation, however, is somewhat different to better fit with the notation used in axiomatic weak memory ordering models. This section can be read independently of § 3.3.

3.4.1 Observing Program Executions

The evaluation of an assertion is based on the observation of an execution of the program under test. An execution is characterized by a series of events and the order in which they are observed.
Program Events

A parallel program comprises a set of threads, each of which generates a series of observable events. Events are generated by instructions executed by the processor, and each instruction may result in a number of events. The execution of the assignment \( x := y + z \), for instance, may give rise to two read and one write events.

I use \( E \) to denote the set of all observable program events. Formally, an event is a tuple comprising a unique identifier \( uid \), the thread identifier \( tid \) of the thread that generated the event, the type of the event, and any data associated with that particular type of event. I distinguish the following types of events:

- **Memory** events \( M \) represent physical memory accesses, such as reads and writes. A memory event is a tuple \((uid, tid, type, \ell, v)\), where \( uid, tid \in \mathbb{N} \) are unique identifiers, \( type \in \{\text{READ, WRITE}\} \) represents the direction of the access, \( \ell \) is the memory location accessed, and \( v \in V \) (where \( V \) is a set of values) is the value read or written. I use \( W_{tid} \ell v \) to denote the write access \((uid, tid, \text{WRITE}, \ell, v)\) and \( R_{tid} \ell \) to denote a read access \((uid, tid, \text{READ}, \ell, v)\) when \( uid \) (and \( v \), respectively) is not relevant in the given context.

- **Fence** (barrier) events \( F \) affect the legal orderings in an execution by enforcing ordering constraints on memory operations issued before and after the fence instruction (c.f. §3.4.1). A fence event is a tuple comprising a unique identifier \( uid \), a thread identifier \( tid \), an architecture dependent fence-type \( type \), and an optional (architecture-dependent) set of ordering constraints (c.f. §3.1). Since these constraints are highly architecture specific, I do not provide a general definition of them here.

- **Scope** events \( S \) are generated upon entry or exit of a syntactic assertion scope. A scope entry event is a tuple \((uid, tid, \text{ENTRY}, \phi_{uid})\), where \( \phi_{uid} \) is an assertion (defined in §3.4.2). A scope exit event is a tuple \((uid, tid, \text{EXIT}, uid_{ENTRY})\),
where $uid_{ENTRY}$ represents the unique entry event corresponding to the scope exit.

- **Skip** events **skip** denote events that are irrelevant for assertion evaluation, such as a no-op instruction.

Note that these events are similar to the events defined in the sequentially consistent semantics (§ 3.3.1), with the addition of **Fence** and **Skip** events.

**Observation Order for Threads**

An observation of a program execution is a sequence of program events. From the viewpoint of a thread $P_n$, each event occurs at a particular point in time (which determines its location in the sequence). In general, there is no global notion of time, and therefore two threads may disagree on the order in which they observe events. This is the fundamental difference between the relaxed and sequentially consistent semantics — the sequentially consistent semantics requires that “All threads must observe the same consistent ordering of events, i.e., there is a single total order” (§ 3.3.1, Requirement 2).

Note that I do not distinguish between “thread-local” and “global” events — conceptually, a thread observes all events in some order, though in practise it is typically infeasible (and unnecessary) to record all observations.

The following definition (borrowed from [22] and consistent with [25]) determines what constitutes an observation of a memory event — from the point of view of a given thread — in terms of the local time of that thread.

**Definition 1** (Observability). A read or write event is observed when the respective memory access takes effect from the point of view of the observer:
• A write to a location in memory is said to be observed by an observer \( P_n \) when a subsequent read of the location by \( P_n \) would return the value written by the write.

• A read of a location in memory is said to be observed by an observer \( P_n \) when a subsequent write to the location by \( P_n \) would have no effect on the value returned by the read.\(^4\)

Fence events have no side effect on the execution other than the constraints they impose on the ordering of events (c.f. §3.4.2 and §3.1). Accordingly, when a fence event is observed is architecture-dependent and determined by the respective ordering constraints. Therefore, I do not provide a general definition. When evaluating a single execution trace, fence events can be treated as skip events. Predictive analysis tools are able to expand a single concrete execution into a family of possible executions which are consistent with the constraints of the original concrete execution, dramatically increasing the bug detection power of a runtime tool. In order to do this expansion, predictive analysis tools require information on the constraints in the original trace — I include fence events as a separate event class because they provide this useful information.

Scope events only affect the thread which generates them (see §3.4.2). Since they are side-effect free, scope events generated by \( P_n \) are observed by \( P_n \) at the point in time when they are generated. All other threads \( P_m \) treat \( P_n \) scope events as skip events.

Observations induce a per-thread total order of events, reflecting the order in which they became visible to a particular thread \( P_n \). I represent this order using an irreflexive transitive relation \( \xrightarrow{\text{obs}_n} : E \times E \). For every pair of events \( e_1, e_2 \) and thread \( P_n \), there is a thread-local observation edge \( e_1 \xrightarrow{\text{obs}_n} e_2 \) iff \( e_1 \) is before \( e_2 \) in this total order.

\(^4\)This definition is not cyclic, since a read observation is defined in terms of the potential effect of a write rather than in terms of the observation of a write.
order, i.e., the thread observes \( e_1 \) before \( e_2 \). I use \( e_1 \xrightarrow{\text{obs}_n} e_2 \) to abbreviate \( \neg(e_1 \xrightarrow{\text{obs}_n} e_2) \) (which implies \( e_2 \xrightarrow{\text{obs}_n} e_1 \) for \( e_1 \neq e_2 \)).

Note that since \( \xrightarrow{\text{obs}_n} \) is irreflexive (i.e., \( e_i \xrightarrow{\text{obs}_n} e_i \)) and transitive, cycles are not allowed. Accordingly, executions characterized by a cyclic ordering relation are in-feasible. While the definition of \( \xrightarrow{\text{obs}_n} \) does not impose any further restriction on executions, the program structure imposes certain restrictions as to the order in which events are generated; the thread executing \( x := y + z \), for example, has to perform reads from \( y \) and \( z \) before it can issue a write to \( x \). I refer to these (thread-local) constraints as the program order \( \xrightarrow{\text{po}_n} : E \times E \).

The program order of a thread \( P_n \) enforces that certain events are observed in a specific order, i.e., \( e_1 \xrightarrow{\text{po}_n} e_2 \Rightarrow e_1 \xrightarrow{\text{obs}_n} e_2 \). The relation \( \xrightarrow{\text{po}_n} \) corresponds to \( \xrightarrow{\text{po}} \) in [1] and to the sequenced-before relation of [14] and is determined by the semantics of the language. The diagram in Figure 3.8, for instance, shows the program order derived from the code fragment \( x = 1; y = x + x \) in the C++ language [49].

In addition to the restrictions imposed by the semantics of the programming language, I require that events are not reordered across assertion scope boundaries. For events \( e_{\text{bef}} \) and \( e_{\text{aft}} \) generated by instructions before and after the beginning of a scope, respectively, and the corresponding scope entry event \( e_{\text{entry}} \), I impose \( e_{\text{bef}} \xrightarrow{\text{po}_n} e_{\text{entry}} \xrightarrow{\text{po}_n} e_{\text{aft}} \) (and similarly for the end of the scope). I emphasize that the relation \( \xrightarrow{\text{obs}_n} \) does not impose ordering constraints on other threads, i.e., \( \xrightarrow{\text{obs}_n} \) is not global in the sense that \( \exists n. e_i \xrightarrow{\text{obs}_n} e_j \) does not imply \( \forall n. e_i \xrightarrow{\text{obs}_n} e_j \).

The underlying memory model, however, may impose ordering constraints across threads. A common assumption is that the memory model guarantees memory co-
Figure 3.9: $P_1$ and $P_3$ observing (a) two instances of $W_2x$ or (b) $R_2x$ and $R_2y$

herence in that for each location, i.e. there is a global total order over the writes to that location (c.f. [II]):

$$
\left( \exists n. \langle uid_1, tid_1, WRITE, \ell, v_1 \rangle \xrightarrow{obs_n} \langle uid_2, tid_2, WRITE, \ell, v_2 \rangle \right) \Rightarrow \left( \forall n. \langle uid_1, tid_1, WRITE, \ell, v_1 \rangle \xrightarrow{obs_n} \langle uid_2, tid_2, WRITE, \ell, v_2 \rangle \right)
$$

(3.2)

Example 3 (Memory Coherence). Fig. 3.9(a) depicts an execution invalidated by the coherence constraint (3.2). Threads $P_1$ and $P_3$ observe the write events $W_2x_1$ and $W_2x_2$ in opposite order. By transitivity, $W_2x_2 \xrightarrow{obs_3} W_2x_1$ follows from $W_2x_2 \xrightarrow{obs_3} R_3x \xrightarrow{obs_3} W_2x_1$. Similarly, $W_2x_1 \xrightarrow{obs_1} R_1x \xrightarrow{obs_3} W_2x_2$ implies that $W_2x_1 \xrightarrow{obs_1} W_2x_2$, and $W_2x_2 \xrightarrow{obs_3} W_2x_1$ follows from (3.2) and $W_2x_2 \xrightarrow{obs_3} W_2x_1$. This contradicts $W_2x_1 \xrightarrow{obs_1} W_2x_2$.

The coherence constraint (3.2) also implies that the observation orders for reads of the same location are consistent for all threads. There is, however, no such constraint for write (or read) accesses to different locations. Fig. 3.9(b), for instance, depicts a valid execution in which the threads $P_1$ and $P_3$ observe two subsequent reads from $x$ and $y$ in opposite order. I point out that this does not contradict the definition in [I] §2.4] that “a read is globally performed as soon as it is performed.” This apparent discrepancy stems from the fact that [I] defines when read events are performed (“the
Figure 3.10: A multi-level cache hierarchy

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>issues read to $\ell$ blocks L3 cache</td>
<td>Cannot affect value in $\ell$ (observes read here)</td>
<td></td>
</tr>
<tr>
<td>reads value of $\ell$ (observes read here)</td>
<td>issues write to $\ell$</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.11: A read by $P_1$ can be observed by $P_3$ before its value has been fixed point when the value of the read is determined” [1, §2.3]) whereas I define when they are observed: it is possible to conceive a cache hierarchy in which thread $P_n$ has already observed a read while thread $P_m$ may still influence its outcome, i.e., according to Definition 1 a read can be observed before it is actually performed.

**Example 4.** The following toy example illustrates the difference between when an event is performed, and when it is observed.

Consider the cache hierarchy in Figure 3.10. Assume in this system that a read or write to a location $\ell$ begins by blocking access to the L3 cache-line associated with $\ell$, then performs the action on the L2 cache and then does whatever other actions are necessary to propagate the value read/written. Since this is a toy example, the practical efficiency of such a scheme is not important.
In this system, if \( P_1 \) issues a read to location \( \ell \), it will first block access to the L3 cache. At this point, no write by processor \( P_3 \) can affect the value read by \( P_1 \), and so \( P_3 \) observes the read as having occurred by Definition 1. Since \( P_2 \) can modify the value without going through the L3 cache, it is still possible for a write by \( P_2 \) to change the value that \( P_1 \) reads. Hence, \( P_2 \) has not yet observed the read. Figure 3.11 shows how this can lead to a read being observed before its value has been fixed.

3.4.2 Operational Semantics of Parallel Assertions

Parallel assertions are evaluated over a given thread’s observation of the program state and execution history. I first provide the syntax and semantics of assertions, and subsequently cover executions.

Structural Operational Semantics for Assertions

Assertions. Each scope event \( e \in S \) has a unique identifier \( uid \) and an assertion expression \( \phi_{uid} \in AExpr \) which follows the grammar specified in § 2.5.3. As in § 2.5.3, I take a structural view of an assertion expression: \( unary-op \) is any unary operator defined in the base language, such as unary negation; \( infix-op \) is any binary operator defined in the base language, such as addition. I use \( lvalues(expr) \subseteq L \) to denote the set of memory locations (\( lvalues \)) whose value is relevant to the evaluation of \( expr \).

Formally, the operator \( lvalues \) is defined inductively as follows:

- If \( lvalues(expr) = L \), then \( lvalues(HO(expr)) = lvalues(unary-op expr) = L \).
- Similarly, \( lvalues(expr_1 infix-op expr_2) = lvalues(expr_1) \cup lvalues(expr_2) \).
- If \( expr \in \{ LR(\ell), LW(\ell), RR(\ell), RW(\ell) \} , \ell \in L \), then \( lvalues(expr) = \emptyset \).
- Finally, \( lvalues(\ell) = \{ \ell \} \) for \( \ell \in L \) and \( lvalues(v) = \emptyset \) for \( v \in V \).

Similarly, \( accessops(expr) \subseteq AExpr \) represents all access events in an expression. Formally, \( accessops(expr) \subseteq AExpr \) is the set of all sub-expressions of \( expr \in AExpr \) of the form \( LR(\ell) \), \( LW(\ell) \), \( RR(\ell) \), or \( RW(\ell) \) (where \( \ell \in L \)).
Figure 3.12: A simple assertion

An assertion set $\alpha$ is a set of tagged assertion expressions $uid: expr$ (where $expr \in AExpr$). Each set $\alpha$ can be partitioned into sets $\alpha|_{uid}$, which denotes the restriction of $\alpha$ to elements tagged with $uid$. The set $\alpha|_{uid}$ itself is inductively defined for each $uid$ as the smallest set satisfying the following rules:

- The assertion $uid: \phi_{uid} \in \alpha|_{uid}$
- Its negation $uid: (\neg \phi_{uid}) \in \alpha|_{uid}$
- If $uid: (\neg expr) \in \alpha|_{uid}$, then $uid: expr \in \alpha|_{uid}$.
- If $uid: (expr_1 \ bop \ expr_2) \in \alpha|_{uid}$, then $uid: expr_1$ and $uid: expr_2$ are in $\alpha|_{uid}$.
- If $uid: (HO(expr)) \in \alpha|_{uid}$, then $uid: expr \in \alpha|_{uid}$

Here, $expr, expr_1, expr_2 \in AExpr$ and $bop$ represents the Boolean connectives supported by the programming language (e.g., $\land$ or $\lor$). Intuitively, $\alpha|_{uid}$ contains the assertion $\phi_{uid}$ and its negation $\neg \phi_{uid}$, as well as all sub-expressions of $\phi_{uid}$. $A$ denotes the set of all conceivable assertion sets.

Example 5. The assertion set $\alpha$ for $!RR(x) || HO(LW(x))$ (from Fig. 3.12) comprises the original assertion as well as the elements $uid: (!RR(x) || HO(LW(x)))$, $uid: !RR(x)$, $uid: RR(x)$, $uid: HO(LW(x))$, and $uid: LW(x)$.

States. A state $\sigma$ is a finite mapping from locations $L$ to values $V$. $S$ denotes the set of all conceivable states. I use the notation $\sigma[\ell]$ to refer to the value of location $\ell$ in the current state, and $\sigma[\ell \mapsto v]$ to denote an update to the state function. In particular, $\sigma[\ell \mapsto v]$ denotes the state that maps $\ell \in L$ to $v \in V$ and all other locations $\ell' \neq \ell$ to $\sigma[\ell']$.

For a given set of locations $L \subseteq L$ the projection $\sigma|_L$ of $\sigma$ to $L$ is the state that maps locations $\ell \in L$ to $\sigma[\ell]$ and is undefined for all other locations.
History. A history $\chi$ is a tuple $\langle \delta, \theta \rangle$ which represents past evaluations of assertions by recording assertion expressions that evaluate to true at some point during the execution. $\chi.\delta$ (of type $AExpr$) represents the immediate past reflecting only the most recent event. $\chi.\theta$ is an assertion set representing the distant past, cumulating all tagged assertions that evaluated to true at some point in the past of the current execution trace. $\mathcal{H}$ denotes the set of all conceivable histories.

Configurations. A configuration $\kappa$ is a tuple $\langle \sigma, \chi, \alpha \rangle$ comprising a state $\sigma \in \mathcal{S}$, a history $\chi \in \mathcal{H}$, and an assertion set $\alpha \in \mathcal{A}$. $\mathcal{C}$ denotes the set of all configurations.

Evaluating Assertions. The value of an assertion is determined by the configuration under which it is evaluated. Given a configuration:

$$\langle \sigma, \chi, \alpha \rangle$$

and the assertion:

$$uid : expr$$

there is a relation:

$$\rightarrow_a \subseteq \mathcal{C} \times (\mathbb{N} \times AExpr) \times (\mathbb{N} \times AExpr)$$

which reduces that assertion. I write

$$\langle \sigma, \chi, \alpha \rangle \rightarrow uid : expr_1 \rightarrow_a uid : expr_2$$

to signify the reduction of $expr_1$ to $expr_2$ under the configuration $\langle \sigma, \chi, \alpha \rangle$.

I define $\rightarrow_a$ recursively as:

1. Assertion expressions (or sub-expressions) that do not contain the operators LR, LW, RR, RW, and HO are evaluated over $\sigma$ according to the semantics of

\footnotetext{\[This notation is inspired by the Dirac delta function $\delta$ and the Heaviside function $\theta$.\]}
the C++ language. Therefore, \( \langle \sigma, \chi, \alpha \rangle \vdash uid: expr \rightarrow_a uid: v \) if \( expr \) evaluates to \( v \in V \) in state \( \sigma \).

2. Expressions involving the access operators LR, LW, RR, or RW are evaluated according to the immediate history \( \chi.\delta \) (where \( T \) is the Boolean value \( \text{true} \) and \( F \) is the Boolean value \( \text{false} \)):

\[
\frac{expr \in \chi.\delta}{\langle \sigma, \chi, \alpha \rangle \vdash uid: expr \rightarrow_a uid: T}
\]

(3.3)

and similarly \( \langle \sigma, \chi, \alpha \rangle \vdash uid: expr \rightarrow_a uid: F \) if \( expr \notin \chi.\delta \).

3. The operator HO maps its parameter \( expr \) to true if \( expr \) evaluates to true in the current configuration, or if \( expr \) was true at some point in the past.

\[
\frac{\langle \sigma, \chi, \alpha \rangle \vdash uid: expr \rightarrow_a (\rightarrow_a)^* \quad uid: T}{\langle \sigma, \chi, \alpha \rangle \vdash uid: HO(expr) \rightarrow_a uid: T}
\]

(3.4)

\[
\frac{\langle \sigma, \chi, \alpha \rangle \vdash uid: expr \rightarrow_a (\rightarrow_a)^* \quad uid: F}{\langle \sigma, \chi, \alpha \rangle \vdash uid: HO(expr) \rightarrow_a uid: b \quad \begin{array}{l} \text{def} \quad T \text{ if } ((uid: expr) \in \chi.\theta) \\ b \text{ otherwise} \end{array}}
\]

(3.5)

Note that the parameters of HO must not be reduced by \( \rightarrow_a \) or evaluated over \( \sigma \) unless this step yields \( T \). This is necessary to avoid mixing values of \( \sigma \) and values of past states during the evaluation.

I denote repeated applications of \( \rightarrow_a \) using the operator \( (\rightarrow_a)^* \), which represents the reflexive transitive closure of \( \rightarrow_a \).
An expression $expr$ is true iff

$$(\sigma, \chi, \alpha) \vdash uid: expr(\rightarrow_a)^* uid: T$$

**Example 6.** The configuration $\kappa \overset{\text{def}}{=} (\sigma, \{LR(x)\}, \{uid : LW(x)\}, \alpha)$ (with $\alpha$ as in Example 5) reflects a recent read access as well as a previous write access to $x$ by the asserting thread. Thus, $\rightarrow_a$ yields $T$ for $HO(LW(x))$ (by rule 3.3, since $\kappa \vdash uid : LW(x) \rightarrow_a uid : F$ and $uid : LW(x) \in \chi.\theta$) and $!F$ for $!RR(x)$ (since $RR(x) \notin \chi.\delta$). The assertion $(!RR(x) \parallel HO(LW(x))$ does not fail at this point.

**Operational Semantics for Events**

**Observable execution traces** A program execution consists of a set of events generated by the various threads. In a sequentially consistent model, these events form a totally ordered sequence, which represents an observable execution trace. As discussed in §3.4.1, the relaxed semantics do not guarantee such a total order; instead, I consider the per-thread observation order represented by the $(\overset{\text{obs}}{\longrightarrow})$ relation. This means that every thread may potentially have a different view of what events are occurring in what order, and hence what the system state is at any given time. Parallel assertions are evaluated from the perspective of the asserting thread.

Formally, an execution of a program $P$ generates a set of events $\mathbb{E}$. An execution trace is an ordered sequence of (potentially some subset) of the events in $\mathbb{E}$. The empty sequence is denoted $\epsilon$. I use the ML-like notation $::$ for sequence concatenation. In ML style, $e_i :: s_j$ refers to a sequence in which the event $e_i$ occurs before the rest of the sequence $s_j$. I also allow sub-sequences to be directly concatenated — $s_i :: s_j$ means that the events in sequence $s_i$ come before the events in sequence $s_j$. The set of all possible observed sequences of events in $\mathbb{E}$, including the empty sequence $\epsilon$, is represented by its Kleene closure $\mathbb{E}^*$. 
I am particularly interested in observable execution traces. A trace of events 
\( ex \in \mathbb{E}^* \) is observable from the point of view of a thread with \( tid = n \) if for every 
sub-sequence \( e_i :: e_{i+1} \) of \( ex \) I have \( e_i \xrightarrow{obs_n} e_{i+1} \).

The effect of an observed event on the configuration The semantics of an 
exteuction is determined by the (reflexive) reduction relation \( \rightarrow \subseteq (\mathbb{E}^* \times \mathcal{C}) \times (\mathbb{E}^* \times \mathcal{C}) \) 
which characterizes the impact of events on configurations. In the following, I define 
this reduction relation \( \rightarrow \) based on \( \rightarrow_a \).

A common side effect of all events is the modification of the history. Let \( \langle \sigma, \chi, \alpha \rangle \) 
be the current configuration and let \( \chi \) and \( \chi' \) denote the history before and after an 
event, respectively. For all events, \( \chi'.\theta \) is \( \chi.\theta \) augmented with all assertion expressions 
in \( \alpha \) that evaluate to true in the previous configuration:

\[
\chi'.\theta \triangleq \chi.\theta \cup \{(\text{uid}:\text{expr}) \in \alpha | \langle \sigma, \chi, \alpha \rangle \vdash (\text{uid}:\text{expr}) (\rightarrow_a)^* \text{uid}:\text{T}\}
\]  

The reduction rules presented in the following refer to \( \chi'.\theta \) as defined in \[3.6\]. 
From now on, I use \( n \) to denote the identifier of the current (asserting) thread.
Figure 3.14: Reduction rules for scope events

1. Fig. 3.13 shows the reduction rules for memory events. I distinguish between events generated by thread $n$ and events generated by other threads in order to determine their effect on the immediate past $\chi, \delta$.

2. Fig. 3.14 shows the reduction rules for scope events. Upon entry of a scope, the respective assertion is added to the assertion set. Note that Rule (3.12) does not allow the system to exit a scope if the corresponding assertion (identified by $eid$) failed. Finally, a thread only observes the scope events generated by itself.

3. Skip and fence events modify the history in accordance with (3.6). In addition, the memory model (see §3.1) may impose ordering constraints on fence events.

Assertion Failures. In case of a failure of an active assertion $\phi_{\text{uid}}$, the configuration can be reduced to a (canonical) error configuration $\text{error}$. I allow this rule to be applied non-deterministically at any point after an assertion failure (but at latest upon exit of the corresponding scope, see Rule (3.12)). This leaves some freedom for
the implementation as to when a failed assertion is reported, as discussed in §3.5.

\[
\begin{align*}
(uid:\neg \phi_{uid}) \in \chi.\theta \lor (uid:\phi_{uid}) \in \alpha \land \kappa \vdash (uid:\phi_{uid})(\rightarrow_a)^* (uid:F)
\end{align*}
\]  

\(\langle ex, \kappa \rangle \rightarrow \text{error} \)  

Example 7. Consider an execution \(W_n x 1 \vdash R_{tid} x\) (where \(tid \neq n\)) starting in the configuration \(\kappa \triangleq \langle \sigma, \{\text{LR}(x)\}, \{uid:\text{LW}(x)\}, \alpha \rangle\) introduced in Example 6. By rule [3.7], I derive \(\langle R_{tid} x, \langle \sigma[x \mapsto 1], \{\text{LW}(x)\}, \{uid:\text{LW}(x)\}, \alpha \rangle \rangle\) from \((W_n x 1 \vdash R_{tid} x, \kappa)\). Note that \(uid:\text{LR}(x)\) is not added to \(\chi'.\theta\) by rule [3.6], since it is not an element of the assertion set \(\alpha\).

3.5  Response to assertion violation

What should happen when a parallel assertion fails depends on the purpose of the parallel assertion system. An implementation may choose when and how to report the error.

3.5.1  Fail-stop

One option is that a parallel assertion should be fail-stop. On a sequentially consistent system, a fail-stop requirement is fairly simple — if an event \(e_t\) with time-stamp \(t\) causes the system to enter a state where any active assertion fails to hold, then the system should immediately stop. No event with time-stamp \(t' > t\) should be executed.

On a relaxed memory system, this requirement must be specified from the point of view of the asserting thread. Assume that the violated assertion occurs on thread \(n\), and was caused by the event \(e_i\). A fail-stop condition means that the runtime system should ensure that there does not exist any event \(e_{i+1}\) in the program such that \(e_i \text{obs}_{n} e_{i+1}\).
A system using parallel assertions to validate a critical system might choose this approach, and enforce this condition through runtime monitoring. Such an approach would potentially introduce significant overhead, because it places assertion evaluation on the critical path for program execution. It is possible that this overhead could be minimized through the use of speculation, although developing such a system is beyond the scope of this dissertation.

3.5.2 Fail before exiting scope

A relaxed condition might be that assertion violations must be reported during the scope of the failed assertion. If assertion $\phi$ on thread $n$ with scope end event $s_{end}$ fails, then the runtime system should ensure that there does not exist any event $e_i$ in the program such that $s_{end} \xrightarrow{\text{obs}_n} e_i$. A system which uses parallel assertions to check a computation before doing IO could use this relaxed condition. There is a trade-off between the latency of the checker in reporting an error, and the efficiency of the checker. Because a relaxed checker is synchronized at the assertion level, not the instruction level, it is less likely to be on the critical path for program execution.

3.5.3 Continue past failure

A third possible condition, useful for debugging, is that any failed assertion should be reported when convenient for the checking implementation. This decouples execution from checking, and allows for full-speed execution. Since errors will potentially be reported long after they occur, an implementation may choose to maintain state associated with detected errors, such as a program counter trace or call stack, to aid the programmer in characterizing the error.

Depending on how severe the consequences of a failed assertion are, allowing program execution to continue after a failed assertion may allow the detection of further bugs in the program, maximizing bugs found per trace.
3.6 Pointers and memory locations

One practical difficulty in implementing both the sequentially consistent and the relaxed ordering models is that the locations upon which an assertion depends may vary during the life of that assertion. For example, consider the following parallel assertion in Figure 3.15, where the value of \( p \) changes during a parallel assertion whose condition references \( p \).

Theoretically, this assertion has a perfectly clear meaning: it asserts that the location \(*p\) is not modified by an external thread. In a tool such as a model checker which has complete knowledge of all program state at all times, this property poses no more difficulty than checking a property where the locations do not vary.

On the other hand, every memory location tracked by a runtime checker increases the overhead of that tool. The dynamic filtering optimization that I present in Chapter 4 relies on pointers having a single static value throughout the execution of an assertion scope. In order to allow my tool to function, I require that any assertion checked by the tool must only reference locations whose addresses do not change.

Figure 3.15: Assertion where memory locations change

```c
void f(int* p) thru {
    doStuff(p);
    p++;
    doOtherStuff(p);
} assert(!RW(*p));
}
```

Figure 3.16: Dynamic runtime interpretation of assertion in Figure 3.15, with implicit checks for memory location stability.
during the assertion scope. This property can optionally be checked by adding the appropriate implicit checks to the assertion condition, as shown in Figure 3.16.

3.7 Related Work

There is a significant amount of active research on the issue of how to formally define semantics for relaxed memory systems. Some of this work focuses on the question of building accurate and precise formal models, while other work focuses on how to debug in the presence of weak memory models.

3.7.1 Formal processor models

Chong and Ishtiaq provide a formal model for the ARM architecture [22]. Owens et al. similarly provide a model for x86 [78]. Alglave et al. developed a Coq model for Power, and provide proofs of correctness for algorithms which restore a given model from a weaker one [1]. My relaxed semantics is specifically designed to integrate well into these models.

3.7.2 Formal language models

A number of languages have formal memory models. Java was one of the early languages to embrace a formal memory model [63], although this model has been shown to block many common optimizations [91].

The current C++ standard [49] is based on the work of Boehm and Adve [14]. Batty et al. provide a rigorous mathematical model of the C++ standard, and prove the correctness of an x86 implementation using that model [5]. As discussed in § 3.2, although my work is inspired by the C/C++ semantics, my approach differs and provides a precise semantics even for programs with data-races.
3.7.3 Debugging relaxed memory model systems

There is a wide variety of work on debugging programs in the presence of weak memory models. One debugging approach, taken by Burckhardt and Musuvathi [16], and Burnim et al. [17], is to monitor a program and detect any non sequentially-consistent executions. RELAXER uses predictive analysis to generate and test relaxed memory model programs, given a sequential trace [18].

Another approach is to minimize the probe effect. Hardware based logging such as Flight Data Recorder allows the observation of events with minimal (2%) perturbation to the program execution on both TSO(x86) and SC systems [101]. A model checker can exhaustively explore all possible interleavings under a given memory model for all possible inputs (e.g. [51]), and hence capture all bugs albeit at a high computational cost. Programs can also be proven correct theoretically. Atig et al. prove that reachability is decidable on both TSO and PSO architectures, but can become undecidable on more relaxed architectures [4].

Trace based analysis can also be used to automatically fix bugs. Liu et al., for example, provide a formal semantics for LLVM bytecode under weak memory models, and show how to add fences to prevent erroneous traces [58].

All of these techniques are orthogonal to my work — they deal with how to generate execution traces, while my work deals with how to interpret a given trace in a weak memory system. Parallel assertions could be implemented as an extension to any of these systems.

3.8 Chapter Summary

This chapter provided a formal definition of parallel assertions, a novel assertion language for detecting intricate concurrency bugs. As Calvin’s evil duplicate realizes, what matters is not the tricky parallel behaviour that is actually occurring; what
matters is what an observer observes. Computer architects follow the same approach, designing systems which allow complicated re-orderings of parallel events.

Since modern processors have complicated memory ordering models (§ 3.1), as do modern programming languages (§ 3.2), I present two related formal semantics.

The first, a sequentially consistent semantics (§ 3.3), is designed to be easy to reason about. It has a single global timeline of events, and a well-defined global state at all times. This model produces debug traces which are simple to understand. It limits the number of interleavings which must be considered by static analysis tools, and hence lowers their overhead.

The second is a relaxed ordering semantics designed to integrate well with weak memory ordering models (§ 3.4). Instead of the single global view of program execution used in the sequentially consistent semantics, the relaxed semantics is based on a per-thread observational view. The relaxed semantics is well suited to runtime checking — as I show in Chapter 4, the relaxed ordering semantics enables several key optimizations in my runtime checker.
Chapter 4

Implementation

This chapter describes **passert**, a debugging tool which implements parallel assertions. I discuss the decision to implement a runtime assertion checker in § 4.1. The architecture of the checker is described in § 4.2. In § 4.3, I describe a set of benchmarks based on the PARSEC benchmark suite (§ 4.3), and in § 4.4 I introduce a series of optimizations that dramatically reduce runtime overhead. I discuss related work in § 4.5 and provide a brief summary in § 4.6.

### 4.1 Fundamental Design Decisions

The first step in implementing an assertion checker is deciding what type of tool to build. My choices during this phase of tool design were informed by my design philosophy: **passert should be useful to real programmers on real programs.** This
philosophy led me to build a software based dynamic runtime checker, which uses a compiler to emit code that generates and checks a decoupled log.

4.1.1 Why a dynamic runtime checker?

Static approaches, such as model-checking, have the advantage that they exhaustively explore the state space of a system, and hence can guarantee to catch every possible assertion violation in a given program. They have the disadvantage that exhaustively exploring the state-space of a system is a PSPACE-complete task \[92\]. As I discuss in the related work section (§ 4.5), there are techniques such as abstraction and intelligent search heuristics that give sub-exponential speed in practise. These techniques often require significant programmer effort, and although they can improve scalability, it remains a major research challenge to use these techniques on large programs.

Dynamic (runtime checking) approaches have the opposite set of trade-offs. A runtime checker can only detect bugs on a given input trace; it cannot tell whether the bug would have occurred with a different input, or a different thread interleaving. Since it is only checking a single execution, a runtime checker can be highly efficient, with linear slowdowns. This means that runtime checking can scale to programs of arbitrary size. To first order, the effective slowdown of runtime checking Microsoft Word should be no different than that of runtime checking Notepad.

Runtime checking is therefore applicable to a wider range of programs, and a wider range of programmer skill, than static techniques are.

4.1.2 Why software based?

A runtime checker must do two things: it must observer the relevant program events as they occur, and it must check those events for correctness.

Hardware based runtime checking techniques, such as Lifeguards, can transparently log events as the program executes. They allow logging and checking to occur
concurrently with program execution, and thus allow execution to continue at high speed \cite{98}. Software based solutions intermingle checking and execution code, significantly slowing program execution. Since there is seldom a perfect correlation between the time necessary to execute an operation and the time necessary to check that operation for correctness, this slowdown can be non-uniform. In a concurrent program, this can cause one thread to be slowed more than others, causing interleavings to manifest that would not have occurred in the original, unmodified program, or eliminating interleavings that would have. This \textit{probe effect} increases the difficulty of reasoning about program execution \cite{66}.

In theory, hardware based loggers are superior because they are less susceptible to this probe effect; in practise, a theoretically powerful hardware solution is useless to a programmer who cannot actually acquire hardware that implements it. A software solution, on the other hand, can be used by any programmer who possesses a suitable computer.

\subsection*{4.1.3 Why compiler based?}

There are a wide range of possible software runtime checking architectures. One possibility is to run the program inside an emulator such as GEMS \cite{65}. This approach effectively allows a hardware logger to be simulated in software, which eliminates the probe effect, but at the cost of extremely high runtime overhead. Another alternative is to use a dynamic rewriting framework such as Pin to add assertion checking code to an existing binary \cite{5}. Dynamic rewriting frameworks have the advantage that they do not require access to the program source-code. This advantage can also be a disadvantage: since dynamic rewriters lack a connection between source-code and executable code, it can be difficult to perform analyses that depend on the structure of the original program.
I chose to use a compiler based approach for two reasons. First of all, a compiler based approach allows the programmer to write assertions directly in the affected program, using the standard syntax discussed in Chapter 2. Secondly, one of the key optimizations requires alias-analysis (§ 4.4.2), and using an existing compiler framework such as LLVM allowed PASSERT to use existing alias analysis tools.

4.1.4 Why a decoupled log?

There are two basic options for an assertion checker — the assertion can be checked synchronously with the execution of the program under test, or the check can be decoupled and performed asynchronously. Synchronous checking allows for bugs to be detected, and the program halted immediately when the assertion failure occurs. However, since assertion checking occurs on the critical path of program execution, synchronous checking can significantly change program timing, which can change the behaviour of a concurrent program. An alternative is to use a lightweight logging mechanism to record relevant events, and then check the log in a decoupled fashion. Since the logging is lightweight, disruption to program execution is limited.

Decoupled checking is also preferable for concurrent programs because an assertion may involve more than one variable. Checking a simple assertion such as \texttt{passert(x>y)} synchronously would require every access to \(x\) to also include an access to \(y\) and vice versa. A program with a decoupled checker would merely need to record both accesses on the log.

4.2 The basic structure of PASSERT

PASSERT is implemented as an extension to the open source LLVM compiler, as shown in Figure 4.1. Its various components are discussed below:
4.2.1 Annotate code with assertions

PASSERT takes as input standard C/C++ code, potentially annotated with parallel assertions. The parallel assertions language (Chapter 2) allows an assertion expression to be any side-effect free expression. Assertions with function calls can be handled by inlining the function body, although this is currently not supported by PASSERT. As I show in Chapter 5, this restriction has minor effects in practise.

At present, PASSERT targets programs using the Pthreads threading library [48]; I expect that it will be easy to extend it to other threading models, such as Microsoft Windows threads [67]. Extending PASSERT to work on more exotic concurrency models such as CUDA [73] or OpenMP [76] is theoretically possible, but may pose implementation challenges and is beyond the scope of this work.

4.2.2 Front-end

PASSERT follows the LLVM architecture, and is split into a front-end, which parses the program and emits LLVM IR (Internal Representation), and a back-end, which performs the optimization and code generation steps. This structural division makes
it easy to extend PASSERT to other languages — since the back-end is language agnostic, only the front-end needs to be replaced to enable parallel assertions for a new language.

The front-end is based on clang 3.0. It parses the input files, replaces assertion begin/end events with function calls, and generates LLVM IR which represents the annotated program.

**Parsing**

The parsing and AST (Abstract Syntax Tree) extensions to clang are relatively straightforward. I modified the clang parser to recognize two new keywords (thru and passert), one new statement thru { ... } passert(expr) based on the do { ... } while(expr) statement, and added new rules to recognize parallel assertion expressions. Instead of adding access predicates as keywords, I treated them as function calls for the purposes of parsing, and converted them (and disallowed other function call types inside assertion expressions) in a later type-checking pass.

I extended the clang AST to include a thru-statement node, based on the existing do-while node, and extended the clang visitor functions to correctly parse this node.

**Code Generation**

Code generation converts the internal clang AST used by the front-end to a format which can be used by later phases of the PASSERT compiler. Most program constructs can be directly converted into standard LLVM IR; those that cannot, such as scope entry/exit events, are represented using opaque functions calls. The use of opaque function calls prevents the LLVM back-end from optimizing around scope entry/exit events, as required by the parallel assertion semantics (Chapter 3). The final result of the code generation pass is legal LLVM IR which can be used without modification by existing LLVM tools.
**Non-assertion code** All AST nodes other than the **thru-statement** are converted into LLVM IR using the standard **clang** code generation functions. The only difference between the IR generated for an instrumented program and the IR generated for the original un-instrumented program is the instrumentation described below.

**Assertion condition** For each assertion condition, **PASSEKT** emits a function which checks whether the assertion holds at a given time.

An assertion condition is a function of a set of values, access predicates, and HO operators, joined by Boolean operators. In my implementation, all values and access predicates needed by a checking function are provided to it as inputs by the runtime system; the runtime system also maintains a Boolean array storing the current value of the HO operators used by each assertion checking function, and passes a pointer to this array to the checking function whenever it is called. The checking function can then update the value of any HO operators that changed as a result of the new event.

Conceptually, each assertion checking function may depend on a different set of variable values and access predicates, and hence each function would have the type signature:

```c
bool f(T1 var1, T2 var2, ..., TN varN,
       bool op1, bool op2, ..., bool opK,
       bool* HO);
```

In practise, the design of the runtime system is significantly simplified if all assertion checker functions have the same type signature, so I required an efficient mechanism to allow functions with variable numbers and sizes of inputs.

The initial design solved this problem by allocating space for each value and access predicate using **malloc**, and then passing the checking function an array of pointers to variables. This mechanism requires a (slow) **malloc** call per variable/access-operator.

```c
bool f(void** values_and_accessops, bool* HO);
```
\begin{verbatim}
assert((i > 5) && !RR(j) || HO(RW(k)));

// translates to:

define i1 @thru. function.CheckingFunc(i64 * nocapture %valArgsArray, i1 * nocapture %hasOccurredArray) nounwind {
    thru.function:
    %0 = bitcast i64 * %valArgsArray to i32 *
    %1 = load i32 * %0, align 4
    %cmp = icmp sgt i32 %1, 5
    %2 = getelementptr i64 * %valArgsArray, i64 1
    %3 = load i64 * %2, align 8
    %lnot = icmp eq i64 %3, 0
    %and = and i1 %cmp, %lnot
    %4 = load i1 * %hasOccurredArray, align 1
    %5 = getelementptr i64 * %valArgsArray, i64 2
    %6 = load i64 * %5, align 8
    %tobool4 = icmp ne i64 %6, 0
    %7 = or i1 %4, %tobool4
    store i1 %7, i1 * %hasOccurredArray, align 1
    %or = or i1 %and, %7
    ret i1 %or
}
\end{verbatim}

Figure 4.2: Code generation for a parallel assertion condition

My optimized assertion checking function design takes advantage of the maximum word size (64 bits) on x86-64 processors. A checking function can take a single input array of 64 bit words, which it can dynamically cast to the correct type as needed. This mechanism requires one malloc per function, saving both time and space.

\begin{verbatim}
bool f(uint64_t* values_and_accessops, bool* HO);
\end{verbatim}

An example of a translated parallel assertion condition is given in Figure 4.2 which shows how the tool handles the different types of assertion sub-expressions: each sub-expression is cast to the correct type, and then the value of the full expression is determined by applying the correct boolean operators to the input values.

The value comparison \( i > 5 \) is handled by lines 7–9. The access predicate \( \text{RR}(j) \) is handled by lines 10–12, and the access predicate \( \text{RW}(k) \) is handled by lines 15-17. The has occurred operator \( \text{HO}(\text{RW}(k)) \) is handled by lines 14–19. The various Boolean connectives are handled by the remaining lines.
typedef int64_t valrst_t;

/* for assertion scope */
typedef bool (*checkingFnType)(valrst_t*, bool);

void beginParallelAssertionScope(void** ptrList, size_t* sizeofList, int* typeList, int num1, int num2, checkingFnType cfnName);

void endParallelAssertionScope(void** ptrList, size_t* sizeofList, int* typeList, int num1, int num2, checkingFnType cfnName);

Figure 4.3: Scope entry and exit function signatures

**thru statement** Code generation of the **thru** statement involves annotating the beginning and end of the scope of the scope with a call to the appropriate runtime function. The arguments to these functions (Figure 4.3) contain all information needed by both the back-end compiler and the runtime system.

- **ptrList[i]** is a pointer to the *ith* location used in the assertion condition. Since the assertion condition may access locations by reference (e.g. `passert(!RW(*p))`), these locations are calculated dynamically, at assertion scope entry, and then stored into the appropriate array. As discussed in §3.6, this creates a requirement that the locations referenced by an assertion not change during the assertion scope; as can be seen by an evaluation of the real-world assertions in Chapter 5, this does not significantly affect the expressiveness of parallel assertions. If the programmer desires, the requirement that locations not change can be added to the assertion and automatically checked — for example, `passert(!RW(*p))` could be rewritten as `passert(!RW(*p) && !RW(p) && !LW(p))`. I currently do not add these implicit properties by default, but instead require the programmer to define correct assertions.

- **sizeofList[i]** is the size of the *ith* location used in the assertion condition. The combination of **ptrList** and **sizeofList** describes the set of memory locations referenced in the assertion condition.

92
• \texttt{typeList[i]} describes how the \textit{i}th assertion location is referenced. Options are \texttt{VALUE,LR,RR,LW,RW}.

• \texttt{num1} is the number of locations referenced in the assertion condition.

• \texttt{num2} is the number of HO operators in the assertion condition

• \texttt{cfnName} is a pointer to the checking function, described earlier in this section.

Currently, the arrays (\texttt{ptrList, sizeofList, typeList}) are allocated before assertion scope entry. De-allocation is handled by the runtime library. One possible optimization would be to statically allocate \texttt{sizeofList} and \texttt{typelist}, since they do not change during program execution.

4.2.3 Back-end

The LLVM based back-end takes the input from the front-end (§ 4.2.2), and compiles it into an executable program. An LLVM execution consists of a series of modular \textit{passes}, all of which share a common \textit{internal representation} (IR). The assertion checking code was implemented as a set of LLVM plugins, with no modifications required to the main LLVM codebase. The back-end execution can be divided into three main phases.

Optimization

The goal of \texttt{ASSERT} is to create an instrumented program that mirrors the behaviour of the original program. I therefore apply the same LLVM optimization passes to the instrumented IR as would have been applied to the original un-instrumented IR. The only difference between the two programs at this phase is that the use of opaque functions to represent scope entry and exit prevents reordering around scope boundaries, as required by parallel assertion semantics (§ 3.3 and § 3.4.1).
**Instrumentation**

Once the program has reached its final, optimized state, I use an LLVM pass to replace selected operations with calls to the corresponding PASSERT runtime functions.

**Program entry/exit**  Program entry and exit are annotated with function calls which allows the runtime to manage the data-structures necessary for program instrumentation.

**Access Events**  Reads and writes which could affect the value of a variable used in a parallel assertion condition are replaced with a call to the corresponding runtime function. As described in § 4.4.2, PASSERT uses an alias analysis pass to detect variable accesses which cannot alias to a parallel assertion. This dramatically reduces the runtime overhead of assertion checking.

**Pthreads function wrappers**  Calls to Pthreads functions are replaced by calls to the corresponding PASSERT runtime wrapper functions.

**Machine code emission**

This phase emits an executable program which can be run by the user. I use the standard LLVM machine code emission pass for this stage.

**4.2.4 Runtime Library**

The runtime library is responsible for responding to relevant program events and determining whether an assertion violation has occurred. The basic architecture of the runtime library is described here; the optimizations described in the following sections can be activated using compile time switches.
Event Logging

For efficiency, I decoupled logging from checking by using a per-thread log and a global time-stamping mechanism. The time-stamp, location, and value of all relevant reads and writes are stored into a thread-local log. Using thread-local logs prevents the log from becoming a serialization point, and the global time-stamping mechanism allows events to be ordered between threads.

Time-Stamping mechanisms  There are two basic challenges that must be solved by a time-stamping mechanism: generating the appropriate time-stamp, and associating the time-stamp with the correct operation (i.e. the correct read, write, or scope entry/exit event).

I investigated a number of different methods for generating time-stamps. RDTSC is a hardware time-stamping mechanism which guarantees a monotonically increasing time-stamp across multiple cores on x86 processors [25]. A shared memory system without such a hardware mechanism could get the same effect in software by using an atomically incremented global variable. The Lamport logical clock is even more flexible, at the cost of increased implementation overhead [56].

There are two basic ways to associate a time-stamp with the associated action. A proof of correctness of these mechanisms, as well as an evaluation of their relative runtime overhead, is given in § 4.4.3.

• Strict time-stamping uses locks to guarantee that a shared counter is incremented atomically with the execution of an event, hence providing a total order over all time-stamped events. In particular, the event logging function acquires a per-variable lock, and then calls a hardware time-stamping mechanism which guarantees atomic access to a monotonically increasing time-stamp across multiple cores. The lock creates an implicit memory fence, and ensures that both the time-stamping and the event action execute as a single atomic unit. The
x86 memory model guarantees a total ordering of lock accesses, so the use of a set of per variable locks reduces lock contention within the logging functions without sacrificing the required ordering properties [25]. Strict time-stamping on other architectures which do not have this property might require the use of more heavy-weight mechanisms.

Similarly, an assertion scope event must be atomic and ordered with respect to the variables upon which it depends. The assertion scope event logging functions therefore acquire the per-variable locks for all variables referenced in the assertion condition. I acquire the locks in sorted order to avoid deadlock [24].

A well-written parallel program will attempt to minimize contention for shared variables, which suggests that there will be a low contention for the locks used in strict time-stamping. It may be possible to explore the use of transactional memory [43] or speculative lock elision [82] to take advantage of this fact and improve performance.

- *Smeared time-stamps* trade precision for performance. There is a single atomic counter $ts$. Scope events atomically increment this counter, for example using the `__sync_fetch_and_add` atomic increment function on x86. All other events $e$ are logged using a preceding and a successive read to the global counter $ts$. The event $(e)$ must not be reordered relative to the counter read $(R_{tid} ts)$. In some cases, this property is guaranteed by the underlying memory model:

\[
(R_{tid} ts) \overset{ppo_{tid}}{\longrightarrow} (e) \overset{ppo_{tid}}{\longrightarrow} (R_{tid} ts)
\]

In cases where it is not, then I ensure the ordering using (non-cumulative) fences if necessary:

\[
(R_{tid} ts) \overset{ab_{tid}}{\longrightarrow} (e) \overset{ab_{tid}}{\longrightarrow} (R_{tid} ts)
\]
In both cases, smeared time-stamping avoids the use of locks.

Since the counter is incremented for every assertion scope event, smeared time-stamping is sufficient to determine whether an event happened within a given parallel assertion scope. A potentially ambiguity may arise in the rare case that different time-stamps are recorded before and after the event. If this difference affects the correctness of an assertion, PASSERT reports a potential error (though I have not observed this in practise).

Log Checker

Checking can either be done online, using a separate checking thread and synchronized queues, or offline, in which case no synchronization needs to be done on the queues. In either case, the checker reads the logs, and merges them into a single time-line. For each event on that time-line, the checker scans the list of currently active parallel assertions. If the event is referenced by the assertion condition, then the checker calls the associated assertion checker function. If the assertion condition checking function returns true, then the assertion holds for that time-stamp. If it returns false, the assertion fails at the time-stamp, and an error is reported.

Response to Assertion Failure  When an assertion fails, PASSERT informs the user and prints out a set of diagnostic information. This information includes which memory access caused the assertion failure (including time, thread_id, value, and type of access), as well as which assertion was triggered. If the user desires, PASSERT can output its full log to a file. If the program has been compiled with debug symbols, it is possible to associate the log information with program locations, although this is not currently implemented. A compiler flag controls whether the executable should abort or continue after an assertion violation.
In addition, PASSERT provides a feedback function which allows user code to block until the checker has evaluated all events before the feedback function call began, and then returns the checker status (i.e. failure or success). The program can use this mechanism to ensure that a dangerous action only occurs after correct execution, or to rollback and recover after an assertion failure. As a convenience, PASSERT can automatically insert a checker feedback call at the end of each thru block.

4.3 Benchmarks

Evaluating the performance of an debugging tool requires a benchmark set of programs. We developed such a benchmark suite by annotating a set of PARSEC benchmarks with appropriate parallel assertions. This set of annotated benchmarks provide a benchmark suite which can be used to analyze the performance of our PASSERT tool.

Note: the applicability study using the University of Michigan collection of concurrency bugs (Chapter 5) was performed as a separate project after the work reported in this chapter.

4.3.1 PARSEC

The “Princeton Application Repository for Shared-Memory Computers” (PARSEC) is a diverse collection of multithreaded programs which has become a de facto standard for architecture and compiler research. It is described in detail in [9], and is available for free download at http://parsec.cs.princeton.edu.

PARSEC has a number of features that make it a good choice for a parallel assertion benchmark suite. Its programs are a representative sample of a wide variety of modern workloads, including scientific, financial, and back-end systems. All of the benchmark programs are available under permissive open source licenses, allowing them to be easily modified to include parallel assertions. For each benchmark, PARSEC provides
a range of carefully designed inputs which can be run on platforms ranging from native hardware to slow timing simulators with minimal loss of fidelity [10].

### 4.3.2 Annotated PARSEC benchmarks

We took a (randomly chosen) selection of PARSEC benchmarks, analyzed them to determine correctness criteria, and then annotated those criteria onto the programs as parallel assertions. This analysis is described in the following section.

**Black Scholes**

*Black Scholes* uses do-all parallelism to calculate the value of a set of options. It was remarkably easy to annotate with assertions — we simply added assertions that privatized variables were neither read nor written by any thread other than their owner.

**Dedup**

*Dedup* compacts a file-system by replacing duplicated sections of files with pointers into a shared hash table. *Dedup* uses a somewhat complicated locking structure. Many concurrent accesses are not directly protected by locks; instead, the code warns through comments that mutexes must be held when calling these unprotected functions. We added assertions to check that these operations proceeded atomically, as specified in the comments.

In addition, elements are connected to mutexes through a hash lookup. We added assertions that any accesses for which a mutex had been acquired should be performed atomically.
Fluidanimate

Fluidanimate is a numerical solver for fluid mechanics. The simulation is divided into a number of chunks, and each thread is responsible for calculating its chunk. We added an assertion that no element is accessed by more than one thread, and were surprised to find that this assertion failed. Examination of the code revealed that the boundaries between chunks are shared between two threads, and can be modified by both. In this case, the use of parallel assertions revealed that our assumptions about the parallelization technique used were incorrect, and helped deepen our understanding of the underlying program. In fact, I believe that this may be a productive way for a programmer to explore the concurrency model of a new program — add assertions that reflect their understanding, and see which ones hold on real executions.

Streamcluster

Streamcluster is a solution to the online clustering problem in data mining. The data points are statically partitioned into thread workloads during the parallel gain computation. The parallel gain computation iterates over a subset of data points in the input data set several times. We added assertions to check that on each iteration where a thread access to a data point has occurred, no other thread accesses that point.

Swaptions

Swaptions uses the Heath-Jarrow-Morton framework to price a portfolio of swaptions. The array of swaptions is divided into a set of blocks, and each block is assigned to a thread. Each thread then uses a Monte-Carlo simulation to determine the cost of the swaption. We added assertions to check that swaptions were correctly privatized, with only one thread reading and writing each element.
4.4 Optimizations and Results

The instrumentation in PASSERT results in a number of side effects. Firstly, logging events takes time, and hence changes the timing behaviour of the program under test. More subtly, the instrumentation adds locks and fences which may rule out executions that are otherwise legal in the underlying memory model. In the following, I discuss three key optimizations that dramatically reduced these effects.

4.4.1 Online Checker Optimization

PASSERT uses a decoupled checking mechanism where events are recorded onto a log which is later checked for assertion violations. I implemented an optimization which executed checking online in parallel with log generation, and compared its performance with an offline checker.

Implementation

Each thread was associated with a thread-local log, which prevented the logs from becoming a serialization point. The online log used a high-performance non-blocking queue; the offline log was implemented using the std::deque library [49]. The online checker was run in a separate thread concurrent with main program execution, while the offline checker was run on the main program thread at program exit.

Avoiding Stalls  An event can only be processed if all events which occur before it in the execution trace have already been processed. If a thread stops generating events, it becomes impossible to determine the correct sequence of events, since the checker has no way of distinguishing between “no event” and “a not-yet-logged event”. The online checker must therefore conservatively wait until the thread resumes generating events. If a thread which is about to stall can generate a Thread Stalled event, the checker can continue without waiting. PASSERT automatically generates such events before
calls to blocking functions such as `pthread_join()`, `pthread_barrier_wait()` and `pthread_cond_wait()`. Programmers writing specialized synchronization libraries can add their own event annotations. They can also insert *Heartbeat Events* into code which is unlikely to generate any logged events, such as calculations on privatized data. As a future extension, I hope to introduce heuristics that will automatically add these events at appropriate points.

**Results**

The performance effects of the online checker optimization (normalized to the runtime of the un-instrumented program) are shown in Figure 4.4. This experiment was performed using dynamic filtering and strict time-stamping. In each case the black bar represents the time spent in the logging phase, and the grey bar represents the time needed by the checker after the logging phase had completed. The combination of the grey and black bars represents the total program execution time.

Online checking requires a more complicated logging mechanism than offline checking, which leads to an increase in the logging phase (black bar) overhead in all benchmarks. On the other hand, online checking reduces effective overhead of checking (grey bar) in all benchmarks. For programs such as *Black Scholes* and *Fluidanimate*, where the checking phase is a significant percentage of total execution time, the online
checker was more efficient than the offline checker; for programs where this was not the case, it was slightly slower.

4.4.2 Filtering Optimization

In this section, I formally prove that only a fraction of the events in an execution is necessary to evaluate an assertion. Since logging information can be expensive, there is a significant optimization opportunity in filtering the executions before they are evaluated. In particular, this means that throughout a scope I can dismiss the events that are irrelevant to the corresponding assertion, as long as assertion failures are preserved:

Definition 2. Parallel assertion equivalent

Let \( ex_1, ex_2 \in E^* \) be two executions delimited by a scope with assertion \( \phi_{uid} \) that contains no other scope events. Then, \( ex_1 \) and \( ex_2 \) are parallel assertion equivalent with respect to \( \phi_{uid} \) iff in all configurations \( \langle \sigma, \chi, \alpha|_{uid} \rangle \) it holds that

\[
\forall \text{error} \left( \langle ex_1, \langle \sigma, \chi, \alpha|_{uid} \rangle \rangle (\rightarrow)^* \text{error} \right) \iff \left( \langle ex_2, \langle \sigma, \chi, \alpha|_{uid} \rangle \rangle (\rightarrow)^* \text{error} \right).
\]

Nested Assertion Scopes. For the purpose of checking whether a specific assertion \( \phi_{uid} \) is violated by an execution, I can treat other scope events similar to \text{skip}. Scope events occurring in threads other than the current one have no impact on the evaluation of \( \phi_{uid} \) (c.f. Rules 3.11, 3.12). A nested scope event only affects the execution if its respective assertion fails. Therefore, it is legal to process assertion scopes independently as long as I report the first assertion that fails (upon exit of the corresponding scope). This allows us to define parallel assertion equivalence (Definition 2) with respect to a single parallel assertion.

In the following, I formally define the projection of configurations and executions to a given assertion, using the \text{lvalues} and \text{accessops} operators defined in § 3.4.2. I then prove that this projection preserves assertion failures.
Definition 3. Projection of a configuration

I define the projection of a configuration \((\sigma, \chi, \alpha)\) to a given assertion \(\phi_{\text{uid}}\) as
\[
(\sigma, \chi, \alpha)|_{\phi_{\text{uid}}} \overset{\text{def}}{=} (\sigma|_{\text{lvalues}(\phi_{\text{uid}})}, (\chi, \delta \cap \text{accessops}(\phi_{\text{uid}}), \chi, \theta, \alpha).
\]

Theorem 1. Projecting a configuration \((\sigma, \chi, \alpha)\) to an assertion \(\phi_{\text{uid}}\) has no impact on the evaluation of \(\phi_{\text{uid}}\). The evaluation of an assertion in a projected configuration is identical to its evaluation in the original configuration:

\[
\forall \kappa \in C. \forall \text{expr}_1 \in \alpha|_{\text{uid}}, \text{expr}_2 \in \text{AExpr}. \\
(\kappa \leftarrow \text{expr}_1 (\rightarrow_a)^* \text{expr}_2) \iff (\kappa|_{\phi_{\text{uid}}} \leftarrow \text{expr}_1 (\rightarrow_a)^* \text{expr}_2)
\]

Proof: By induction on the structure and height of derivations generated by the reduction rules in §3.4.2.

Definition 4. Projection of an execution

The projection of an execution \(e \in E^*\) to an assertion \(\phi_{\text{uid}}\) is defined inductively by \(e|_{\phi_{\text{uid}}} \overset{\text{def}}{=} \epsilon\) and \((e :: e)|_{\phi_{\text{uid}}} \overset{\text{def}}{=} e|_{\phi_{\text{uid}}} :: (e|_{\phi_{\text{uid}}})\), where

\[
e|_{\phi_{\text{uid}}} \overset{\text{def}}{=} \left\{ \begin{array}{ll}
e & \text{if } (e = W_{\text{tid}} \ell v) \\
(\ell \in \text{lvalues}(\phi_{\text{uid}})) \lor \\
(\text{tid} = n \land \text{LW}(\ell) \in \text{accessops}(\phi_{\text{uid}})) \lor \\
(\text{tid} \neq n \land \text{RW}(\ell) \in \text{accessops}(\phi_{\text{uid}})) & \text{or } (e = R_{\text{tid}} \ell v) \\
\text{skip} & \text{otherwise}
\end{array} \right.
\]
Theorem 2. Let \( \text{ex} \in \mathbb{E}^* \) be an execution that is delimited by a scope associated with \( \phi_{uid} \) and does not contain any other assertion scopes. Then \( \text{ex} \) and \( \text{ex}|_{\phi_{uid}} \) are parallel assertion equivalent with respect to the assertion \( \phi_{uid} \).

Proof: By induction on the length of \( \text{ex} \). The base case is trivial, since \( \epsilon|_{\phi_{uid}} = \epsilon \).

For all other cases, whether \( \text{error} \) is reachable depends on whether \( uid : \phi_{uid}(\rightarrow_{a})^* uid : \mathbb{F} \) in any configuration during the execution. This follows from Rule 3.14 and the fact that all reduction rules incorporate \( \chi^\prime, \theta \) (3.6). Accordingly, it is (by virtue of Theorem 1) sufficient to show that for all non-scope events \( e \) the following claim holds:

\[
\forall (\sigma, \chi, \alpha|_{\phi_{uid}}). \left( \left( \{ e :: ex, \sigma, \chi, \alpha|_{\phi_{uid}} \} \rightarrow (ex, \langle \sigma^\prime, \chi^\prime, \alpha|_{\phi_{uid}} \rangle) \Rightarrow \langle ex, \langle \sigma^\prime, \chi^\prime, \alpha|_{\phi_{uid}} \rangle \rangle \right) \right)
\]

I perform a case split over the relevant rules. Assume that \( e = W_{tid} \ell v \). Then the claim is trivially true if \( e = e|_{\phi_{uid}} \). If \( e|_{\phi_{uid}} = \text{skip} \) we have \( \ell \notin lvalues(\phi_{uid}) \), and therefore \( (\sigma[\ell \mapsto v]|_{lvalues(\phi_{uid})} = \sigma|_{lvalues(\phi_{uid})} \). Moreover, \( (tid = n) \Rightarrow LW(\ell) \notin accessops(\phi_{uid}) \) and \( (tid = n) \Rightarrow RW(\ell) \notin accessops(\phi_{uid}) \), and therefore \( \{ LW(\ell) \} \cap accessops(\phi_{uid}) = \emptyset \) and \( \{ RW(\ell) \} \cap accessops(\phi_{uid}) = \emptyset \), which establishes the claim for Rules 3.7 and 3.9 respectively.

A similar argument applies for the case that \( e = R_{tid} \ell v \) (addressing Rules 3.8 and 3.10 respectively). The claim holds trivially for \( \text{skip} \) and fence events. \( \blacksquare \)

The proof of Theorem 2 shows that events can be treated as \( \text{skip} \) as long they have no impact on the evaluation of \( \phi_{uid} \) (in accordance with Theorem 1).

Theorems 1 and 2 enable the following optimization. For each scope instance, no events need to be logged before its respective entry event. Upon reaching a scope entry event with assertion \( \phi_{uid} \), my implementation records only the relevant fraction \( \sigma|_{\phi_{uid}} \) of the state. After that, it is sufficient to log all events \( e|_{\phi_{uid}} \) (in observation order) until the corresponding scope exit event is reached.
Implementation

To counteract the probe effect, I implemented an event filter that conservatively approximates the set of events required by Theorem 2.

Firstly, the filter uses an alias analysis to generate a conservative approximation to $\sigma_{\phi_{uid}}$. If the alias analysis determines that a given memory location will never be referenced by any parallel assertion, then by Theorem 2 there is no need to log any events involving that memory location. This significantly reduces the number of instructions that need to be instrumented and hence significantly reduces runtime overhead.

Secondly, I use dynamic filtering to determine at runtime which locations are not referenced by any live assertions, and hence need not be logged. I implement this dynamic filter using a hash-table which records whether accesses to a given memory location need to be logged. Collisions in the hash-table may cause unnecessary logging, but will never cause an event to be missed from the log.

This further reduces – but does not entirely eliminate – the probe effect.

Results

The effect of the filtering optimizations on the annotated PARSEC benchmarks described in § 4.3 are shown in Figure 4. In order to limit the overhead of running PASSERT with filtering disabled, the experiment was performed using the reduced input sizes available in PARSEC. Since reduced-sized PARSEC inputs are designed to mimic the behaviour of full-sized inputs, this should not significantly affect the validity of the comparative results.

The “No Filter” bar represents the total number of events, and hence the total number of events that would need to be logged given no filtering optimizations. The “Static Filter” bar shows the number of events that need to be logged after the static alias analysis pass. The “Dynamic Filter” bar represents the number of events that
require logging after both the dynamic and static filters are applied. The static filtering optimization reduces the number of logged events by a factor of $4.3\times$ on average. The dynamic filtering optimization is dramatically more effective, reducing the number of logged events by *four orders of magnitude* on average.

Filtering effectively *enables* runtime checking. Before this optimization, most instrumented programs simply ran out of memory, while all of the benchmarks completed successfully after filtering.

### 4.4.3 Time-stamping Mechanisms

Under certain conditions it is sufficient to approximate the observation order $\text{obs}_n$ by a partial order. In particular, I show that for a certain class of assertions all that matters is the order of events with respect to the scope events $S$.

**Definition 5.** *Assertion violating event patterns*
For a given parallel assertion $\phi_{uid}$ with assertion expression $aexpr$, there is a set of events/configurations that can cause that assertion to be violated. Assertion violating event patterns represent this set in a concise manner.

An assertion violating event pattern is a tuple $\langle \text{type}, \ell, \text{thread}, f(v, \kappa) \rangle$ where type $\in \{\text{READ, WRITE}\}$ represents the direction of the access, $\ell$ is the memory location accessed, thread $\in \{\text{LOCAL, REMOTE}\}$ represents whether the access is local or remote, $v \in \mathbb{V}$ (where $\mathbb{V}$ is a set of values) is the value read or written, $\kappa$ represents the configuration at the time the event occurred, and $f(v, \kappa)$ is a Boolean function over values and configurations.$^{[1]}$ I write $\ast$ to represent any unnecessary values in the tuple.

$^{[1]}$This is similar to the the syntax for observation events in §3.4.1.
For example:

- The assertion \texttt{passert(!RW(x))} is violated by a write to \texttt{x}, regardless of the configuration or value.
  \[ \langle \text{WRITE}, x, \text{REMOTE}, f(*,*) \rangle, \text{where } f := \top \]

- The assertion \texttt{passert(x == 4)} is violated by a write to \texttt{x} with a value other than 4, regardless of the configuration.
  \[ \langle \text{WRITE}, x, *, f(v,*) \rangle, \text{where } f := (v \neq 4) \]

- The assertion \texttt{passert(x > y)} is violated by a write to \texttt{x} with a value less than the value of \texttt{y} in the current configuration \(\kappa\), or by a write to \texttt{y} with a value greater than or equal to the value of \texttt{x} in the current configuration \(\kappa\).
  \[ \langle \text{WRITE}, x, *, f(v,\kappa) \rangle \text{ or } \langle \text{WRITE}, y, *, f'(v,\kappa) \rangle \]
  where:
  \[
  f := v > \kappa.\sigma[y] \\
  f' := v \leq \kappa.\sigma[x]
  \]

Notice that each assertion violating event pattern \(\mathcal{P}_i\) directly corresponds to a simple parallel assertion expression \(expr_i\), which is true iff the parallel assertion violating event pattern occurs. For example, \(\langle \text{WRITE}, x, \text{REMOTE}, f(*,*) \rangle\) is equivalent to \texttt{passert(!RW(x))}.

The mapping between \(\mathcal{P}_i\) and \(expr_i\) for patterns which do not depend on \(\kappa\) is given in Figure 4.6. My experience in Chapter 5 suggests that the most common expressions involving access operators \{LR, LW, RR, RW\} have \(f(v) = \text{true}\), in which case the first four cases in Figure 4.6 simplify to the appropriate values shown in the third column.
Definition 6. **Normalized parallel assertion form**

For every assertion \( \phi_{uid} \), there is an assertion violating event pattern set \( P \) corresponding to that assertion, and vice-versa. I define a parallel assertion as being in normalized form i\( f \) all every term \( expr_i \) in the assertion \( \phi_{uid} \) is in one to one correspondence with a pattern \( P_i \) in \( P \) (i.e. is in the form):

\[
passert(!\text{expr}_1 \lor \text{expr}_2 \lor \ldots \lor \text{expr}_n))
\]

or equivalently, by DeMorgan's law:

\[
passert(!\text{expr}_1 \land !\text{expr}_2 \land \ldots \land !\text{expr}_3))
\]

Note that a finite length assertion expression can only reference a finite number of \textit{lvalues}. Each \textit{lvalue} can be covered by (at most) four assertion violating event pattern for appropriate choices of \( f_i \):

\[
\{\text{READ}, \ell, LOCAL, f_{LR}(v, \kappa)\}
\]

\[
\{\text{WRITE}, \ell, LOCAL, f_{LW}(v, \kappa)\}
\]

\[
\{\text{READ}, \ell, REMOTE, f_{RR}(v, \kappa)\}
\]

\[
\{\text{WRITE}, \ell, REMOTE, f_{RW}(v, \kappa)\}
\]

This implies that the size of \( P \) is finite for a finite sized assertion expression.

Definition 7. **Permutation independent assertion**

Let \( \phi_{uid} \) be an assertion, \( ex \in E^* \) be a possible execution for that assertion, and \( \pi(ex) \) be an arbitrary permutation of \( ex \). I say that \( \phi_{uid} \) is permutation independent i\( f \) all \( ex \) and all permutations \( \pi(ex) \), \( ex \) and \( \pi(ex) \) are parallel assertion equivalent.

Theorem 3. An assertion \( \phi_{uid} \) is permutation independent iff there exists a set \( P \) of assertion violating event patterns associated with \( \phi_{uid} \), none of which reference the configuration \( \kappa \) — i.e. every event pattern in \( P \) can be written in the form \((\text{type}, \ell, f(v, *))\).
Proof:

• If: I prove that the order of the sequence is irrelevant by showing that the following logical equivalence holds:

\[
\left( \langle ex, (\sigma, \chi, \alpha|_{uid} ) \rangle \rightarrow^{*} \text{error} \right) \iff \exists e \in ex. \forall (\sigma', \chi', \alpha|_{uid} ). \langle e \vdash e, (\sigma', \chi', \alpha|_{uid} ) \rangle \rightarrow^{*} \text{error}
\]

(\Leftarrow): The implication holds trivially in this direction.

(\Rightarrow): I need to show that \( \phi_{uid} \) does not depend on the configuration before the execution of \( e \in ex \). Let \( (\sigma'', \chi'', \alpha|_{uid}) \) be the configuration after the execution of \( e \). By Theorem 1,

\[
\langle \sigma'', \chi'', \alpha|_{uid} \rangle \rightarrow_{a}^{*} \text{error}
\]

if

\[
\langle \sigma'', \chi'', \alpha|_{uid} \rangle |_{\phi_{uid}} \rightarrow_{a}^{*} \text{error}
\]

By Definition 5, an event \( e \) will only lead to the \text{error} state iff it matches some pattern \( p \in P \). By definition, there exists \( P \) in which none of the assertion violating event patterns reference the configuration, so this property trivially holds.

• Only if: Assume that there does not exist such a set \( P \). Then there is at least one event \( e \) that will lead to an error if it occurs in configuration \( \kappa_1 \), but will not lead to an error if it occurs in configuration \( \kappa_2 \). Let \( s \) be a sequence of events that modifies the configuration from \( \kappa_1 \) to \( \kappa_2 \). Then the execution \( s \vdash e \) is not parallel assertion equivalent with \( e \vdash s \), and hence \( \phi_{uid} \) is not permutation independent.

\[ \blacksquare \]
Lemma 1. An assertion $\phi_{uid}$ is permutation independent iff it can be expressed in the following form, where $op$ is an operator from the set $\{LR, LW, RR, RW\}$ and $expr(\ell)$ represents an expression of the form $op(\ell) \land f(\ell)$, as shown in Figure 4.6:

$$!(expr_1(\ell_1) \lor expr_2(\ell_2) \lor \ldots \lor expr_k(\ell_k))$$

or equivalently, by DeMorgan’s Law:

$$\neg expr_1(\ell_1) \land \neg expr_2(\ell_2) \land \ldots \land \neg expr_k(\ell_k)$$

Note that in the common case, where access operators and functions are not mixed, this expression simplifies to

$$!(op_1(\ell_1) \lor \ldots \lor op_k(\ell_k) \lor f_{k+1}(\ell_{k+1}) \lor \ldots \lor f_m(\ell_m))$$

Proof: By Theorem 3 an assertion is permutation independent iff there exists a set $P$ of assertion violating event patterns associated with $\phi_{uid}$, none of which reference the configuration $\kappa$. An assertion violating event pattern which does not reference the configuration $\kappa$ can be written in normal form:

$$\text{passert}(\neg (expr_1 || expr_2 || \ldots || expr_n))$$

where $expr$ is in the form shown in Figure 4.6.

The conditions for $\phi_{uid}$ in Corollary 1 are defined based on the structure of $\phi_{uid}$, and hence hold for any program. For a particular program, it may be that certain events/configurations can never happen in any feasible execution. Criteria which exploit this fact (based on partial order reduction, for instance) may allow for more aggressive optimizations. Note that in the most extreme case — if it can be established statically that the assertion holds — it is not necessary to log any events at all. In
general, an approach based in this insight is obviously impractical. However, since observing events and orderings between events is expensive, an improved relaxation function which limits the number of required observations can significantly reduce the work required by the checker.

**Implementation**

Assertion evaluation requires recording the timing of remote events relative to the asserting thread. For events generated by different threads this requires a certain amount of synchronization. My implementation uses a global time-stamp for this purpose.

A time-stamping mechanism must correctly associate events with time-stamps reflecting the observation order. In a weak memory model, this may need to be enforced through the use of locks and memory fences. Consequently, stronger ordering requirements exacerbate the probe effect, which in turn may rule out otherwise legal executions. In Fig. 3.4, for example, a time-stamping mechanism inserting a fence between $W_1 x_1$ and $W_1 y_1$ effectively prevents reordering of these events, thus hiding the bug (as explained in Example 1).

By Corollary 1, the order of events within a scope can be safely ignored in certain cases (such as the program in Example 1). My experience suggests that this relaxation is admissible for many assertions: 14 out of 17 bugs in the University of Michigan Collection of Concurrency Bugs can be captured using parallel assertions; all of those assertions are amenable to relaxed timing (Chapter 5). This observation led me to implement two different time-stamping mechanisms described previously in § 4.2.4.

*Strict time-stamping* associates every relevant program action with a unique time-stamp which enforces a total ordering of events. *Smeared time-stamps* associates each event with an epoch counter which only imposes an order relative to scope begin/end events.
Results

In order to measure the run-time overhead, I evaluated the runtime overhead of PASSERT by annotating a set of PARSEC benchmarks with parallel assertions. We did not discover any new bugs in this widely used benchmark suite.

The effect of the relaxed timing optimization are shown in Figure 4.7. Strict timing had an overhead of 6.6×, which can be reduced to 3.5× through the use of smeared time-stamps.

Threats to validity  This experiment shows that relaxed timing improves the performance of a runtime checker; it does not speak to whether it improves the bug finding power of such a checker. Memory ordering bugs clearly occur in theory: Lamport’s description of sequential consistency contained a simple mutual exclusion algorithm that only works correctly on a sequentially consistent machine [55]. Such bugs would be caught by an ideal implementation of relaxed timing parallel assertions.

In practise, determining the practical bug-finding power finding power of relaxed timing parallel assertions requires answering the subquestions: How often do memory ordering bugs appear in practise? Given that these bugs depend on precise re-orderings between adjacent variables, how likely are they to occur given the extra overhead imposed by a (relaxed timing) runtime checker? The main focus of this
project was on improving performance; the evaluation of the effectiveness on weak memory bugs remains future work.

4.5 Related Work

Just as there is a trade-off in property specification between completeness and programmer effort, there is a trade-off in property checking between completeness and checker effort. These two trade-offs are orthogonal — it is possible to check whether a complicated temporal logic property holds on a single trace, and it is possible to run a theorem prover to find all possible data-races in a program.

At the one extreme, some tools are trace based. They efficiently monitor single runs of a program. This limits their observational power — they can only find bugs that manifest in the observed execution trace. However, such tools can be fast. Vlachos et al. present a hardware assisted runtime checker for parallel computations that can check for misuse of tainted data with low overhead [98].

Trace based techniques can be extended in several ways. Trace based tools can use predictive analysis to determine whether an error could have occurred on a given trace, even if it did not actually do so due to non-deterministic scheduling [34].

Tools such as DMP [29] and CHESS [70] perturb the execution of the program under test in ways that they hope will cause more bugs to manifest.

Model checkers, such as SPIN, can explore the transition relation of a program, effectively testing all possible execution traces at once [47]. Although in theory, a model checker may have to consider all possible thread interleavings over all possible traces, in practise search heuristics often allow for more efficient execution. STORM reports good results using bounded model checking to explore all execution traces that have less than $k$ context switches [54].
Theorem provers such as HAVOC use contract based reasoning to abstract complex code [6]. Ideally, if the code is modular and the contracts are well defined, HAVOC can factor a large program as a set of smaller, more tractable programs. HAVOC has been used to prove the correctness of the synchronization protocol in a 300,000 line Microsoft Windows component.

My work adds a unique point to this spectrum by providing flexible user defined assertions that cover a range of concurrent programming bugs. While I describe checking parallel assertions on a single trace, either on or off-line, these assertions could also be verified over all traces through model checking/theorem proving.

4.6 Chapter Summary

This chapter presented PASSERT, a practical runtime checker for parallel assertions. PASSERT was designed to be useful to real programmers on real programs. This design goal led me to build PASSERT as a compiler based runtime checker: if a programmer can compile and run their program using a standard compiler, they can (with minor additional effort) compile and run their program using PASSERT. Calvin’s “duplicator” leveraged the power of the existing “transmogrifier” technology; PASSERT extends the power of the LLVM compiler framework, and hence offers a programmer all the features of a full-powered modern compiler. More powerful parallel assertion checking tools such as a model checkers are possible, but they require significantly more programmer expertise to use, and do not scale well to large programs.

Runtime checking can impose significant overhead on the execution of the program under test. PASSERT uses a number of provably correct techniques to reduce this overhead.

First of all, PASSERT decouples logging from checking. A runtime checker must observe the set of relevant events, and must check those events for correctness.
PASSERT’s decoupled logging framework limits the effect of the checking phase on program execution.

Secondly, PASSERT uses both dynamic and static filters to limit the number of events that must be logged, and hence the overhead of both logging and checking.

Finally, PASSERT takes advantage of the power of the relaxed memory semantics described in Chapter 3 to implement smeared time-stamping. Smeared time-stamping dramatically reduces the need for mutexes in the logger, which both improves runtime efficiency, and allows more (potentially buggy) executions to manifest.

Unlike Calvin’s tool, PASSERT will never directly print money for programmers. However, its combination of ease of use and reasonable overhead suggest that it may be useful for the more prosaic task of finding concurrent bugs.
Chapter 5

Real World Applicability

The true test of an assertion language is its effectiveness at detecting and diagnosing real bugs in real programs. I evaluated the effectiveness of parallel assertions by attempting to apply them to an independently selected collection of 17 real world concurrency bugs from major open source programs. In the majority of cases, parallel assertions were able to capture the cause of these real world bugs without any modification to the underlying source code. In a few cases, the bugs were beyond the scope of parallel assertions, and adding effective assertions would require significant code rewriting. I propose extensions to the parallel assertion language to help cover these bugs.
5.1 Benchmark selection

One of the key challenges in performing a study such as this is selecting a representative set of bugs to analyze. A good bug collection should be independently selected. It should include a range of types of bugs from different types of programs. The bug reports should be clear and accurate, with relevant code sections and interleavings clearly indicated. The University of Michigan Collection of Concurrency Bugs scored highly on these selection categories.

5.1.1 The Michigan Collection of Concurrency Bugs

The University of Michigan Collection of Concurrency Bugs is an independently selected collection of 17 real world concurrency bugs from major open source programs. The collection was originally developed as a benchmark for the bug-resistant multicore architecture described in [102], and is currently publicly available on the web at http://www.eecs.umich.edu/~jieyu/bugs.html. Each bug in the collection includes an English description of the bug, keywords describing the type of bug, a code snippet showing the buggy code, an example buggy interleaving, and the steps necessary to reproduce the bug.

Each bug in the collection is labelled with a description of the bug type. These descriptions are somewhat informal, and often overlap. In particular, the line between data-race and atomicity is not clearly defined. Many bugs are described as containing both data-races and atomicity violations, and MySQL #169 is simply described as “a concurrency bug that involves multiple variables”. Based on these descriptions, I divided the bugs in the collection into four types (data race, atomicity, order violation, or deadlock) depending on the underlying cause of the error. Bugs where synchronization was used incorrectly were categorized as “atomicity”; bugs where synchronization was missing entirely were categorized as “data race”.

119
5.1.2 Representativeness of the Michigan Collection

The Michigan collection is representative in terms of the types of bugs included — it includes data race, atomicity, order violation, and deadlock bugs. The proportions of the bugs are broadly in line with those suggested by previous research, although the number of deadlock bugs (6%) is low compared to the 31% reported by Lu et al. 

It is less representative in the types of programs: with the exception of Transmission, a multi-threaded BitTorrent client, the collection focuses on server applications, and does not include GUI user programs such OpenOffice or Mozilla.

5.2 Study design

For each bug in the University of Michigan bug collection, I attempted to write a parallel assertion to detect its symptoms and diagnose its underlying cause.

5.2.1 Characterization of assertion effectiveness

Assertions have two purposes: to detect unexpected events that may represent bugs, and to test hypotheses to diagnose the cause of these bugs.

Detect bug symptoms

First of all, assertions can detect the symptoms of a bug. For example, consider the buggy function in Figure 5.1. This function uses a globalCount variable to store how many active instances of buggyFunction() are currently executing. Since the C/C++ increment instruction is not atomic, it is possible that two simultaneous increments to globalCount could lead to only a single increment action, as discussed in § 2.4.2. If the corresponding two decrement instructions are executed correctly, globalCount will go negative, which represents an error. An assertion which detects this condition, such as the one in Figure 5.2, allows programmers to detect that the program has a
bug. However, the root cause of the bug may be separated in both space and time from its visible effect, and so the assertion does not necessarily allow the programmer to immediately diagnose and fix the bug.

**Diagnose bug causes**

The second use for assertions is to *diagnose the cause* of a program bug. Figure 5.3 gives an example of such an assertion. This assertion will fail at the time, and at the program location, where the actual error occurred, allowing the programmer to correctly determine the precise cause of the error, and fix their code.

### 5.3 Data race bugs

#### 5.3.1 Apache #21287

This is a fairly subtle bug caused by a misunderstanding about the semantics of a library function. The programmer of the `decrement_refcount()` function used the atomic function `apr_atomic_dec()` to decrement a refcount, and cleanup the object if the refcount was zero. As shown in Figure 5.4, the programmer’s expectations for this function did not match the actual function’s behaviour. Although the update is atomic, a data-race can occur between the update on line 14 and the read on line 16. This race allows the double-free bug shown in Figure 5.5.

**Parallel Assertion**

This bug could be detected and diagnosed using the simple parallel assertion shown in Figure 5.6.
void buggyFunction()
{
    globalCount++;//should have been synchronized
doStuff();
globalCount--;//should have been synchronized
}

Figure 5.1: Simple example of buggy program

void buggyFunction()
{
    thru {
        globalCount++;//should have been synchronized
doStuff();
globalCount--;//should have been synchronized
    } passert(globalCount >= 0);
}

Figure 5.2: Assertion to detect symptom of bug in Figure 5.1

void buggyFunction()
{
    thru {
        globalCount++;//should have been synchronized
doStuff();
globalCount--;//should have been synchronized
    } passert(!RW(globalCount));
doStuff();
    thru {
        globalCount--;//should have been synchronized
    } passert(!RW(globalCount));
}

Figure 5.3: Assertion to diagnose cause of bug in Figure 5.1
//programmer expected:
int apr_atomic_dec(int* loc)
{
    atomic{
        *loc--;
        return *loc;
    }
}

//On some architectures:
int apr_atomic_dec(int* loc)
{
    atomic{
        *loc--;
    }
    return *loc;
}

Figure 5.4: Expected vs. actual behaviour for apr-atomic-dec

Thread 1

decrement_refcount(...) {
    apr_atomic_dec(&obj->refcount);
}

Thread 2

decrement_refcount(...) {
    apr_atomic_dec(&obj->refcount);
    if (!obj->refcount) {
        cleanup_cache_object(obj);
    }
    if (!obj->refcount) {
        cleanup_cache_object(obj);
    }
}

Figure 5.5: Apache Bug #21287

thru {
    apr_atomic_dec(&obj->refcount);
} passert(!RW(&obj->refcount));

Figure 5.6: Parallel assertion to find Apache Bug #21287

123
5.3.2 Apache #25520

This bug is caused by an unsynchronized write to a log. The description given by the University of Michigan team is somewhat confusing — they imply that the error is due to contention on the `buf->outcount` variable. While it is possible that this variable could be corrupted, the more likely error (discussed in the Apache bug report) is that the `memcpy` on lines 25 and 33 of Figure 5.7 will conflict. This is a data-race bug because there is no synchronization in the function.

Parallel Assertion

This bug can be both detected and diagnosed with a simple parallel assertion that ensures that no other thread makes a conflicting write to the buffer, as shown in Figure 5.7

5.3.3 Cherokee Bug 1

This bug is almost identical to Apache #25520 (§ 5.3.2). Two unsynchronized writes occur to a string buffer, leading to data corruption.

Parallel Assertion

The parallel assertion in Figure 5.8 would both detect the bug, and allow its diagnosis.

5.3.4 MySQL #644

This bug is due to a data race between the calculation of a data-value and its use. As shown in Figure 5.9, between the calculation of the number of fields that need to be used in the query (i.e., have `field->query_id == thd->query_id`) on line 8 and the use of this value on line 14, the underlying data can change, causing the loop
apr_status_t
ap_buffered_log_writer(request_rec *r,
    void *handle,
    const char **strs,
    int *strl,
    int nelts,
    apr_size_t len)
{
    char *str;
    char *s;
    int i;
    apr_status_t rv;
    buffered_log *buf = (buffered_log*)handle;

    thru {
        if (len + buf->outcnt > LOG_BUFSIZE) {
            flush_log(buf);
        }

        if (len >= LOG_BUFSIZE) {
            apr_size_t w;
            str = apr_palloc(r->pool, len + 1);
            for (i = 0, s = str; i < nelts; ++i) {
                memcpy(s, strs[i], strl[i]);
                s += strl[i];
            }
            w = len;
            rv = apr_file_write(buf->handle, str, &w);
        }
        else {
            for (i = 0, s = &buf->outbuf[buf->outcnt]; i < nelts; ++i) {
                memcpy(s, strs[i], strl[i]);
                s += strl[i];
            }
            buf->outcnt += len;
            rv = APR_SUCCESS;
        }
    passert(!RW(*buf));
    return rv;
}

Figure 5.7: Apache #25520
to terminate incorrectly. This is a data-race bug because there is no synchronization
protecting any of the variables involved.

Parallel Assertion

Conceptually, this bug is fairly simply to capture and diagnose — simply add a parallel
assertion that there is no remote write to the \texttt{tables[i].table->field->query_id}
field, as shown in Figure 5.9, line 23. However, this property must hold over a \textit{collection}
of fields, and it is difficult to express such a collection in the current \texttt{PASSERT}
implementation. A programmer could try to finesse this problem by manually “un-
rolling the loop” as they write their assertion expression.

Possible language improvements

A \texttt{foreach} operator in the language would significantly reduce programmer work in
this example, as shown in Figure 5.10.

5.3.5 MySQL \#791

MySQL 4.0.12 had a bug that occurs during log rotation which can cause updates
to be missed from the log. The \texttt{MYSQL\_LOG::new\_file()} command first closes the
old log file, then opens a new file. The programmer’s intention was that this should complete as a single atomic unit. However, as shown in Figure 5.11, a second thread can read the temporary log_type = LOG_CLOSED value, and fail to log an event.

Parallel Assertion

The parallel assertion in Figure 5.11 would both detect the bug, and allow for its diagnosis. This example shows the importance of the RR operator.
Figure 5.10: Use of `foreach` on MySQL #644

Figure 5.11: MySQL #791

128
This bug is caused by a data-race between the check that a log file is active, and the use of that log file. During this time, another thread can come in and rotate the log, inactivating the currently active file. This bug is a mirror image of MySQL #791, where a log file was erroneously detected to be open before it was ready to use; in this case, a currently valid log file can be closed while it is in use.

5.3.7 Parallel Assertion

This bug can be both detected and diagnosed by wrapping the if statement with an assertion that its condition does not change during the execution of this statement.
See § 5.3.8 for discussion of a possible language improvement that would make this condition easier to write.

5.3.8 MySQL #3596

This bug is an extremely simple data-race bug, almost identical to the bug in § 2.4.1. A thread checks that a pointer is non-null, then de-references the pointer. Since the check and the use are not atomic, an invalid pointer de-reference can occur. This example, like bug MySQL #2011 (§ 5.3.6), occurs because the condition of an if statement is assumed to remain constant throughout the body of the then block, whereas, in fact, a data-race can cause it to change.

Parallel Assertion

This bug could be both detected and diagnosed by the assertion in Figure 5.13.

Possible language improvements

Since bugs caused by the invalidation of an if condition during the then block appear to be a common pattern, these examples suggest there may be value in having a new macro thruIf(cond) which would be syntactic sugar for the expression in Figure 5.14.
```c
thruIf(cond) => if(cond) {
    thru {
        then-body;
    } passert(cond);
} else {
    thru {
        else-body;
    } passert(!cond);
}
```

**Figure 5.14:** thruIf(cond) operator

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
</table>
| void ...print_thd(..) | bool do_command(...) {
| thruIf (thd->proc_info) {
| putc(‘ ’, f); | thd->proc_info = 0; |
| fputs(thd->proc_info, f); |
| } |
| } |

**Figure 5.15:** Parallel assertion for MySQL #3596 using thruIf construct

Figure 5.15 shows how the thruIf statement would simplify the parallel assertion for MySQL #3596.

### 5.3.9 MySQL #12848

MySQL maintains a global cache of SQL queries, whose size is recorded in the variable query_cache_size. This bug occurs because the resize() function updates query_cache_size to the new cache size before the cache has actually been updated. A second thread can then read this invalid cache size, and make an invalid access to the cache.


Parallel Assertion

The assertion in Figure 5.16 captures this unexpected read, allowing diagnosis of the bug.

5.4 Atomicity Bugs

5.4.1 Aget Bug 2

When a user terminates an aget download by pressing `ctrl-c`, the program logs both per-thread and total statistics. Ideally, the total statistics should be identical to the sum of the per-thread statistics. Unfortunately, the variable `bwritten`, which records the number of bytes written, is not updated in a proper atomic section. This can cause the final output of the program to be invalid.

The buggy interleaving is shown in Figure 5.17. Although this program contains a data-race due to the unsynchronized access to `bwritten` on line 13 in addition to the
atomicity violation, protecting this access with `bwritten_mutex` would not prevent the bug.

**Parallel Assertion**

The bug can be both detected and diagnosed using a parallel assertion which detects when a remote thread writes `bwritten`, as shown in Figure 5.17.

### 5.4.2 Apache #45605

Apache uses a set of listener and worker threads, coordinated using queues. The `queue_info->idlers` variable stores whether there is work waiting for the listener thread. If `queue_info->idlers` is non-zero, the listener thread takes an element off the queue; if it is zero, the thread waits for a signal from a worker thread that there is new work to process. The problem is that because the broadcast of a signal is not atomic with the action that is being signalled, the property may be invalid by the time the signal is received. The correct method for using signals is to wrap the wait inside a while loop:
while(!property){
    cond_wait(signal);
}

This bug occurs because Apache uses an if statement on line 19 of Figure 5.18 instead of the correct while statement. As can be seen in Figure 5.18, by the time the signal is received, queue_info->idlers may be zero, leading the listener thread to attempt to dequeue an item from an empty queue.

**Parallel assertion**

The bug can be detected using the assertion shown in Figure 5.18 lines 732 which detects when the queue size becomes invalid. However, an assertion failure associated with the decrement on line 30 would not necessarily lead the programmer to deduce that the bug was caused by a missing while loop on line 19.

The parallel assertion on lines 2729 would allow the programmer to detect that the number of idlers was unexpectedly zero after the signal was received, which would hopefully allow them to diagnose the problem with the missing while statement. Interestingly, in this case the bug is not within the scope of the thru statement that is meant to diagnose the bug.

### 5.4.3 memcached #127

This bug is another example of the difficulty of using caches correctly in a multi-threaded environment. In this bug, shown in Figure 5.19 Thread 1 retrieves an item from the cache (Line 6), and then operates on it using the add_delta() function (Line 19). Although add_delta() operates atomically on the object, it is possible for another thread to replace the object in the hash-table (Line 15) as it is being updated, causing Thread 1’s update to be lost.
Initial: queue_info->idler == 0;

Listener Thread

listener_thread (...) {
    thru {
        ...}
        
        if (queue_info->idlers == 0) {
            ...}
            
            queue_info->idlers--; 
            
            if (queue_info->idlers == 0) {
                lock(idlers_mutex);
                ...}
                
                if (queue_info->idlers == 0) {
                    cond_wait(wait_for_idler);
                    lock(idlers_mutex);
                    cond_signal(wait_for_idler);
                    unlock(idlers_mutex);
                }
                unlock(idlers_mutex);
            } }
            
            thru {
                ...}
                
                }passert(queue_info->idlers >0);
                queue_info->idlers--; 
                
                }passert(queue_info->idlers >= 0); 

Worker Thread

ap_queue_info_set_idle (...) {
    {
        ...}
        
        prev_idlers =
        queue_info->idlers;
        queue_info->idlers++; 
        ...}

Figure 5.18: Apache #45605
Parallel Assertion

Since removing an item from the hash-table requires modifying its `it_flags` field, I can detect and diagnose this bug with a simple parallel assertion that this field is not remotely modified.

5.4.4 MySQL #169

MySQL-3.23.56 had a concurrency bug which caused it to produce a nonsensical log: for example, it could report a successful insert before the associated table had been created. The programmer intended for the table deletion, and the insertion of the new data, to be a single atomic operation with no other interleaving writes. However, as Figure 5.20 shows, the locks used did not cover the entire desired atomic region,
causing an atomicity violation. Note that this bug is not a data-race, since all accesses to the log are protected by a lock.

**Parallel Assertion**

The parallel assertion shown in Figure 5.20 tests whether the log order represents the actual order of events, i.e. do an operation (such as creating a table) and the logging of that operation form a single atomic unit. This seemingly simple test requires the expressiveness of parallel assertions. It would be incorrect to mark the entire `generate_table()` function as atomic, because it correctly accesses shared variables in a non-transactional way. In addition, while conflicting writes to the logger represent an error, reads may not. This assertion captures these subtleties, and successfully diagnoses the cause of the error.
5.4.5 MySQL #12228

This bug is a subtle error in the way that MySQL handles stored routines. To reduce contention, each thread maintains a separate stored routine cache. MySQL also maintains a global variable cversion which is updated whenever a stored procedure is created/deleted/modified.

When a shared procedure is called, MySQL checks whether the local cache c is up-to-date by checking whether c.version < cversion. If the cache is up-to-date, the stored procedure is fetched from the cache and executed. If it is not, the cache is flushed and rebuilt.

A subtle problem occurs because stored procedures can recursively call other stored procedures. If cversion remains constant, everything works correctly. If cversion is incremented between the calls to the parent and the child procedure, then the child procedure will flush the stored procedure cache, deleting the parent procedure. When the parent resumes execution, unexpected behaviour may result.

Note that this is not a data-race, since all accesses to cversion are protected by a lock.

Parallel Assertion

This bug can be easily detected and diagnosed using a parallel assertion that checks whether the cversion field has been unexpectedly updated, as shown in Figure 5.21.

5.4.6 Apache #21285

I quote from the description given in the bug report at http://web.eecs.umich.edu/~jieyu/bugs/apache-21285/DESCRIPTION:

The bug is in mod_mem_cache module, which is used to cache dynamic objects. The module works as follows: the dynamically generated objects
bool mysql_execute_command(THD *thd) {

... switch (lex->sql_command) {
  ...
  case SQLCOM_CALL:
    thru{
      sp_head *sp;
      ...
      if (... || open_and_lock_tables(thd, ...))
        goto error;
      ...
      if (!(sp = sp_find_procedure(thd, lex->spname, TRUE))) {
        ...
        goto error;
      } else {
        ...
        res = sp->execute_procedure(thd, ...);
        ...
      }
    }
passert(!RW(cversion));
    break;
  ...
  case SQLCOM_DROP_PROCEDURE:
    sp_head *sp;
    ...
    sp = sp_find_procedure(thd, lex->spname);
    ...
    if (sp) {
      ...
      result = sp_drop_procedure(thd, lex->spname);
    }
    ...
    break;
  ...
}

Figure 5.21: MySQL #12228
are first inserted in the cache as temporary objects with a default size (in function \texttt{create_entity()}). Later, the object will be resized once the actual size is available. To resize the temporary object, the module first removes it from the hash table and then re-inserts it with its correct size (in function \texttt{write_body()}). The temporary object is supposed to be in the cache when resizing happens. However, it is possible that the temporary object is removed from the cache due to capacity reason (in function \texttt{cache_insert()}). If that happens, the size of the cache could be a negative number and finally crash the server.

An example buggy interleaving is shown in Figure 5.22. The fix to the program is extremely simple — the Apache team wrapped the call \texttt{cache_remove()} on line 24 with a check that the object remains within the cache, avoiding the double-free bug.

\textbf{Parallel Assertion}

The final symptom of the bug (that the cache size becomes negative) is easily detectable by wrapping the program with a parallel assertion that the cache size remains non-negative. The programmer could then further narrow down the possible location of the bug by verifying that the object is in the cache before removing it on line 24 (finding the actual fix implemented by the Apache team).

The underlying problem with this code, however, is the lack of atomicity between the two steps of cache insertion. Inserting a cache item ought to be an atomic operation, but is instead split into two operations across two different functions. This underlying bug is difficult to diagnose using parallel assertions because the error occurs across program scopes. There is no single syntactic scope that is associated with the desired atomic region, and hence there is no good place to place the parallel assertion scope.
int create_entity(...) {

... lock(sconf->lock);
cache_insert(..., obj);
unlock(sconf->lock);

... lock(sconf->lock);

void cache_insert(...) {

obj = cache_pq_pop(c->pq);

... cache_hash_set(...); // remove from hash table
c->current_size -= c->size_entry(obj);

c->free_entry(obj);

... unlock(sconf->lock);

... lock(sconf->lock);

... unlock(sconf->lock);

... lock(sconf->lock);

... cache_remove(..., obj);

... cache_insert(..., obj);

... unlock(sconf->lock);

...}

Figure 5.22: Apache #21285

Possible language improvements

These types of non-syntactic atomicity bugs could be represented by allowing the programmer to begin and end scopes dynamically, instead of requiring assertion scopes to map directly to syntactic program scopes. This maps well to the current PASSERT implementation, where scope begin and end are represented as calls to runtime library functions. These calls could be directly exposed to the programmer as an alternative API for using parallel assertions. Syntactically scoped parallel assertions guarantee that every scope entry is matched by exactly one scope exit; dynamically scoped parallel assertions would require the programmer to ensure this property.
Figure 5.23 shows an example of how dynamic scoping might detect the bug Apache #21285. The assertion is dynamically triggered by the call to `beginPassert` on line 8 and remains active until the corresponding call to `endPassert` on line 25. Note that this occurs on a different syntactic scope, and hence cannot be easily expressed using a standard parallel assertion. As written, this example requires checking a function `cache_contains`, which is permitted by the parallel assertion language, although not currently supported in `passert` for performance reasons. If `free_entry(obj)` modifies `obj`, it would be possible to simply assert `!RW(*obj)`; otherwise, ghost state could be added to track whether objects were cached.

5.5 Order violation bugs

5.5.1 pbzip2 0.9.4

A `pbzip2` execution has a master thread, a set of consumer threads which read and compress the uncompressed input files, and an output thread which outputs the compressed result. The programmer assumed that when the output thread was completed, the program was completed, and therefore it was safe to delete unneeded data-structures before exiting the program. As shown in Figure 5.24, it is possible that a consumer thread may still be waiting to finish, and will use a deleted data-structure (line 23).

Parallel Assertion

This bug is difficult to detect and diagnose using parallel assertions. There is no easy way in the parallel assertions language for a thread to assert that the variables it is working on have not been deleted. Depending on the exact implementation of the delete function, it is possible that there may be unexpected remote writes to the deleted memory, but this is not guaranteed to occur.
Even if these remote writes were detected, the cause of the bug lies in the incorrect join on line 16. Diagnosing the error would require detecting this, which parallel assertions do not provide an obvious mechanism for.

### 5.5.2 Transmission 1.42

This bug is a simple ordering violation, where a variable is read before it has been initialized. The programmer was aware of the possible race condition, and attempted to prevent it with a 500ms delay before calling `allocateBandwidth()`. While this
Thread 1

```c
void main(...) {
    ...

    for (;;) {
        pthread_mutex_lock(fifo->mut);
        while (fifo->empty) {
            if (allDone == 1) {
                ...
            }
            producer(..., fifo);
            fifo->empty = 1;
            ...
        }
        queueDelete(fifo);
        fifo = NULL;
        pthread_mutex_unlock(fifo->mut);
        return NULL;
    }
}
```

Thread 2

```c
void *consumer(void *q) {
    ...
    for (;;) {
        pthread_mutex_lock(fifo->mut);
        while (fifo->empty) {
            if (allDone == 1) {
                ...
            }
        }
}
```

// start reading in data
producer(..., fifo);

// wait until exit of thread
// should have joined all threads
// not only the output thread
pthread_join(output, NULL);

// reclaim memory
queueDelete(fifo);

Figure 5.24: pbzip2-0.9.4

delay increases the probability of the correct ordering occurring, it does not guarantee it.

Parallel Assertion

The bug can both detected and diagnosed with the simple assertion shown in Figure 5.25. Interestingly, this assertion was not included in [88], where I reported that this bug could not be easily detected using parallel assertions.
5.6 Deadlock bugs

5.6.1 Aget Bug1

This bug is a deadlock bug caused by asynchronous thread cancellation. When a download completes, aget uses the `pthread-cancel()` function to terminate the now unneeded signal handling thread. If the cancelled thread was in a critical section, the locks that it held will never be released, causing a deadlock when the main thread attempts to acquire those locks. An example of such a buggy interleaving is shown in Figure 5.26.

Parallel Assertion

Parallel assertions are not designed to detect deadlock bugs. There does not appear to be any obvious assertion that could detect this bug.

5.7 Chapter Summary

Programmers writing concurrent programs are much like Calvin in the strip that begins this chapter. As long as all their concurrent objects cooperate properly, every-
thing works great. But when something goes wrong, as it all too often does, detecting and fixing the cause of the problem can be a major struggle.

This chapter evaluates the effectiveness of parallel assertions for detecting and fixing real concurrency bugs. In particular, I analyzed 17 real-world bugs from the University of Michigan Collection of Concurrency Bugs. For each, I attempted to write an assertion which would ideally capture both the symptoms and the cause of the concurrent bug.

The results are extremely promising: almost all (15/17) of the bugs in the University of Michigan Collection can be detected using parallel assertions, as shown in Figure 5.27. Of these, fourteen can also be diagnosed using parallel assertions; one (Apache #21285) is a multi-function atomicity violation, which would be difficult to capture in a single syntactic scope. PASSERT is designed to express safety properties; it is not designed to detect liveness properties such as deadlock.
5.7.1 Possible language improvements

My experience with adding assertions to detect/diagnose these bugs suggests some possible language extensions that would improve both the ease of writing assertions, and their expressiveness.

- MySQL #644 (§ 5.3.4) suggests the value of a `foreach` operator to allow programmers to make statements about collections of objects.

- MySQL #2011 (§ 5.3.6) and MySQL #3596 (§ 5.3.8) both suggest the utility of a `thruIf` statement.

- Apache #21285 (§ 5.4.6) shows that some parallel programs require atomic regions that do not map to syntactic program scopes. Extending parallel assertions to allow assertions would make it easier to detect and diagnose bugs in these programs. In this case, it would be the responsibility of the programmer to ensure that scope entry and exit functions were properly paired.

5.7.2 User-defined granularity

One of the most powerful features of parallel assertions is that they let the programmer precisely define the granularity at which they want their program to be checked. Some data-races represent real bugs, while others are intentional and benign. Similarly, a programmer who wishes to ensure that certain actions are atomic may not require that all actions in the region are atomic. As we saw in the case of MySQL #169 (§ 5.4.4), parallel assertions can capture these distinctions, and hence precisely capture bugs without false alarms.
<table>
<thead>
<tr>
<th>Program</th>
<th>Bug Type</th>
<th>Detect Symptom</th>
<th>Diagnose Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aget Bug2</td>
<td>Data Race</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Apache #21287</td>
<td>Data Race</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Apache #25520</td>
<td>Data Race</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Cherokee Bug1</td>
<td>Data Race</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MySQL #644</td>
<td>Data Race</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MySQL #791</td>
<td>Data Race</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MySQL #2011</td>
<td>Data Race</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MySQL #3596</td>
<td>Data Race</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MySQL #12848</td>
<td>Data Race</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Apache #45605</td>
<td>Atomicity</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Memcached #127</td>
<td>Atomicity</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MySQL #169</td>
<td>Atomicity</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MySQL #12228</td>
<td>Atomicity</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Apache #21285</td>
<td>Atomicity</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Pbzip2 0.9.4</td>
<td>Order Violation</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Transmission 1.42</td>
<td>Order Violation</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Aget Bug1</td>
<td>Deadlock</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Figure 5.27: Summary of parallel assertion bug coverage
Chapter 6

Conclusions

6.1 Future work

This thesis opens many avenues for future work.

6.1.1 Improvements to PASSERT

One important area of future work is engineering improvements to PASSERT. The current implementation of PASSERT is an engineering prototype — it is sufficient to explore the effectiveness of parallel assertions, but it is not yet a polished product.
Better debugging messages

The current implementation of passert generates user messages that directly reflect its internal data-structures. For example, when an assertion failure occurs, passert prints a fairly daunting wall of text representing, among other things, the (numeric) thread_id of the offending thread, a (numeric) token representing the assertion, and the memory location whose access caused the assertion failure. This information is useful to the debugging programmer, but is not terribly user-friendly.

Fortunately, more user-friendly information already exists in the form of debugging symbols, which relate numeric memory locations to user-friendly program locations. It would be an extremely useful engineering task to integrate this information into passert. I could then present user messages like:

`***Error:
*a write of value "4" to location "foo" by thread "2"
*at line "23" in function "bar"
*if (foo == 0)
* foo = 4; <= offending line
*violated an assertion on lines "12-23" in function "baz"
*on thread "1"
* unlock(l);
*passert(!RW(foo) && !RR(bam)); <= violated assertion`

Integrate into other debugging tools

One commonly used debugging tool is interactive debuggers such as gdb, which allow the programmer to interact with a running program [93]. For example, a programmer using such a tool might set break points, and then single-step through the function, inspecting the value of program variables to look for unexpected conditions. Although these tools are generally effective for debugging sequential programs, they have se-
rious limitations in the parallel context. Integrating parallel assertions into such a framework would have several advantages.

First of all, the logging framework of PASSERT provides significant information about the execution of a parallel program, including a trace of the execution of relevant events. In addition, parallel assertions provide a powerful way to describe conditions on the execution of a parallel program. It would be very interesting to explore the use of parallel assertion like conditions as breakpoints.

Improve performance

Profiling and tuning code  As discussed in Chapter 4, PASSERT has an overhead of approximately $3.5\times$ in the most highly optimized case. This compares favourably with tools such as Pin, which has order of magnitude slowdowns for comparable “heavyweight” analysis tasks [5]. However, it is clearly not suitable for everyday use in a production environment.

The implementation of PASSERT described in this thesis is a research prototype. It was developed in accordance with good software engineering principles, but time and manpower resource constraints led me to focus on functionality and clarity of code, rather than concentrating on aggressive optimization. For example, although I attempted to lay out memory in order to limit cache traffic, I did not do an exhaustive study of program memory layout. There are almost certainly significant opportunities to further tune PASSERT.

Turning off logging when not in use  A significant proportion of program execution time occurs when no parallel assertions are active [81]. PASSERT uses a dynamic filter to attempt to reduce overhead in this case — however, it cannot currently remove the overhead caused by the filter itself. This turns a simple memory access into, at minimum, a hash-table lookup followed by a conditional jump. Initial explorations
suggest that this penalty can be dramatically reduced by appropriate hardware \cite{81}. However, such hardware does not currently exist, and is not likely to exist in the near future.

Instead, it might be possible to use a runtime system to dynamically swap between an instrumented and an un-instrumented version of the executing program. The active instrumentation could be tailored to the active parallel assertions.

**Explore new architectural features**

As I discussed in §4.2.4, the use of architectural techniques such as *transactional memory* \cite{43} and *speculative lock elision* \cite{82} could significantly improve the overhead of \texttt{passert} logging. Processors implementing these techniques are beginning to become commercially available, and so exploring these possibilities is now feasible.

### 6.1.2 Alternative Implementations

**Hardware runtime checking**

On a modern processor, an access to a shared variable will typically lead to the production of *cache coherence messages*. With minor modifications to the cache coherence protocols (for example, to transmit messages reflecting assertion scope events), it should be possible to detect parallel assertion violations from a log of such messages. Unlike software logging, such a hardware runtime checker would have minimal effect on the timing and behaviour of the program under test. To my knowledge, no current commercially available architecture offers the ability to log the necessary events, but this possibility could be explored using an architectural simulator such as GEMS \cite{65}.
Static Analysis

There has been significant work on using static tools to analyze potential parallel execution traces. Work in this area includes theorem provers, model checkers, and predictive analysis tools. Typically, these tools have solved the state explosion problem by abstracting the problem. One fruitful area for future research would be to explore the abstractions best suited to detecting parallel assertion violations.

6.1.3 Support for other programming languages/models

My explorations of parallel assertions have mostly focused on the C/C++ family of languages — my syntax, semantics, and tool are given in terms of a C/C++ implementation. However, parallel assertions are suited to any imperative multi-threaded language, and in fact may have advantages on some other languages.

Java

Java is a strongly typed object oriented language that typically compiled to bytecode which is executed inside a runtime environment. These features are all potentially useful to a parallel assertion implementation.

The strong type system enforced by Java could improve the effectiveness of both the static and dynamic checker. Since two Java items can only alias if they have compatible types, alias analysis in Java should be more precise than in C/C++. Unlike in C/C++, where any location can potentially be modified by any function, the strict object-orientation in Java means that there is a limited set of functions that can modify a given location, and hence a limited number of places where logging needs to be inserted.

Java also supports synchronized blocks, which ensure that only a single thread acts on a given object at a time. From the point of view of a parallel assertion logger,
it may be possible to consider such blocks as atomic sections, and simply report the
net actions taken by such a block, reducing log traffic.

Finally, as discussed above in §6.1.1, much of the logging overhead occurs when
no parallel assertions are live, and is hence unnecessary. Java’s powerful runtime
system is already capable of doing dynamic recompilation to reduce the overhead of
commonly used code segments. It seems likely that it would be possible to use these
features to reduce logging overhead by dynamically swapping between logging and
non-logging code.

GPUs

One of the major developments in modern computer architecture has been the rise
of Graphical Processing Unit (GPU) based computing [77]. GPUs offer incredible
amounts of concurrent resources, but also require programmers to write their pro-
grams in precise ways that that advantage of these resources. The current lack of
observability into program execution, and the difficulty of writing correct GPU pro-
grams, create a significant opportunity for a debugging system like parallel assertions.
On the other hand, the massively parallel nature of GPU computation, and the rel-
ative imbalance between computing power and I/O bandwidth on GPUs will require
a significant redesign of any runtime logging system for GPUs.

6.1.4 Better benchmarks

As discussed in Chapter 4, my current performance benchmark suite is based on
PARSEC. Although PARSEC is designed to be a representative suite of parallel pro-
grams, it is by its nature an incomplete selection of programs. I hope to improve this
benchmark selection in three ways: First, by including our annotated programs based
on the University of Michigan bug collection, as discussed in Chapter 5. Secondly, I
hope to include other major benchmarks such as SPLASH-2 [100]. Finally, by adding a
collection of real-world memory-model-specific bugs, since none of the current benchmarks target these bugs. Adding a benchmark with memory-model-specific bugs would allow for more accurate evaluation of the effectiveness of the relaxed timing optimization.

6.1.5 Fault localization

Detecting assertion violations is only the first step in the debugging process. Once the bug has been detected, the cause must be determined before the bug can finally be fixed.

Deterministic replay

One of the major problems in parallel bug localization is the “Heisenbug” effect — bugs appear in a particular interleaving, but cannot be replicated in subsequent tests because of the non-deterministic nature of multi-threaded computation. There are existing tools which allow programmers to repeatedly replay an execution until they find the bug [29]. It is likely possible to integrate PASSERT with one of these tools, and hence allow a programmer to replay a program where an assertion failed and find the root cause of the failure.

Static analysis

Another technique for fault localization is to use the power of modern SAT solvers and model checkers to automatically isolate the root causes of the bug. Given an execution trace and a failing assertion, a model checker can be used to determine which statements within the program could have been changed to prevent the assertion from being activated. The set of such possible statements represents a set of possible locations for the bug.
6.1.6 Usability studies

Parallel assertions are designed to be simple and easy for programmers to use. My experience, and that of my collaborators, provides anecdotal evidence for the utility of parallel assertions. It would be extremely useful to perform rigorous usability testing measuring the relative ability of programmers to find bugs with and without the use of parallel assertions.

6.1.7 Automatic assertion generation

One challenging, but potentially useful addition to parallel assertions would be an automatic assertion generation tool. There are a number of possibilities for how such a tool could be designed.

One possibility is to automatically annotate common code constructs with corresponding parallel assertions. For example, as discussed in §5.3.8 bugs caused the invalidation of an if condition during the then block appears to be a common pattern. A tool could easily add a parallel assertion condition to catch bugs of this type.

Another common source of bugs is incorrect usage of locks. Programmers often attempt to ensure that atomicity of a code segment using locks. If these locks are used incorrectly, atomicity is violated and bugs can occur. An automatic assertion generation tool could add parallel assertions that check for the atomicity of lock-protected code sections.

In addition to looking for bug patterns in code, an automatic tool might detect invariants in correct program executions. These invariants can then be used as assertions. This technique has already proven effective at finding bugs in both parallel and sequential programs [33, 38].
6.2 Final thoughts on parallel assertions

I began this thesis by asking whether it was possible to develop a parallel debugging tool which was simple, powerful, and efficient. This dissertation describes my work in developing and implementing a parallel debugging tool which meets these criteria.

6.2.1 Are parallel assertions simple to use?

Yes.

As discussed in Chapter 2, parallel assertions are a simple extension to existing languages such as C/C++. A programmer needs to learn one new statement (thru/passert), and five new operators (LR, LW, RR, RW, HO).

Parallel assertions are also conceptually simple. The mathematical complexity of the semantics described in Chapter 3 is a result of the complexity of the memory ordering models implemented by modern processors. In particular, a sequential assertion is a Boolean expression on the system state at a point in time. Parallel assertions are similar, but also add the currently observed action to the system state. This allows programmers to write simple Boolean expressions that depend on the actions taken by threads. Thus, although the assertion implementer must deal with the complexity of the parallel assertion semantics, the task of the assertion user remains relatively simple.

Parallel assertions can be easily checked at runtime using PASSERT. Using the PASSERT compiler requires the programmer to:

1. Add assertions to their program
2. Modify their makefile to call PASSERT instead of LLVM
3. Link the PASSERT runtime library to the generated executable
4. Execute the new program as normal

In general, a programmer who can use a compiler can use PASSERT.
6.2.2 Are parallel assertions powerful enough to capture real bugs?

Yes.

Parallel assertions are powerful enough to detect and diagnose real bugs in real programs. As I showed in Chapter 5, parallel assertions were able to detect fourteen out of the seventeen concurrency bugs from the major open-source programs analyzed. Parallel assertions were further able to diagnose all but one of the detected bugs. Further work could extend parallel assertions to enable them to capture the missing bugs — possible language improvements that would extend the effectiveness of parallel assertions are discussed in §5.7.

Bugs that would be masked by a sequentially consistent implementation can be detected using the weak memory model semantics discussed in §3.4.

False positives present a major problem of parallel debugging tools. Tools which make it difficult to precisely express programmer intent can require the programmer to wade through a sea of false positives to find the one true bug. For example, a data-race detector uses an implicit specification which may detect benign data-races. Similarly, an atomicity checker requires the programmer to denote atomic blocks. Real programs often have blocks that are atomic for some variables, but do not require full atomicity for all variables. Detecting an atomicity violation in this case would violate programmer intent.

Parallel assertions allow programmers to precisely declare their expectations about program behaviour. Any failure of a parallel assertion therefore represents a mismatch between program specification and implementation. In some cases, the problem may lie with the programmer’s expectation; in others, with the implementation. In all cases, however, the programmer learns something useful from the parallel assertion failure.
6.2.3 Are parallel assertions efficient?

It depends on the context.

Efficiency can be measured by the degree to which a debugging tool slows down the program it is checking. In the context of attempting to debug a program error, the $3.5\times$ overhead of `passert` (Chapter 4) compares favourably with other runtime systems such as Valgrind, which have seen wide adoption as debugging tools.

Further improvements in efficiency would enable always-on use of parallel assertions. For example, it might be possible to use statistical sampling to speed up assertion checking. Modern hardware is beginning to support transactional memory, which might lower the overhead of time-stamping mechanisms, particularly in the relaxed timing context. Another possibility that has had encouraging initial results is to use user direct messaging to selectively enable and disable event logging [61, 84]. Always-on checking would allow the detection and diagnosis of parallel bugs as they happen, and would significantly speed the debugging process.

6.2.4 Can we help ordinary programmers debug parallel programs?

Yes.

The advent of the multi-core era has created a need for parallel programs, and hence for effective parallel debugging tools. This dissertation has addressed this problem by introducing a novel assertion language called parallel assertions. This new approach allows programmers to precisely state their assumptions about the execution of a parallel program, and to test these assumptions as the program executes. Parallel assertions provide programmers with a tool which is simple, powerful and efficient, and which will help to bridge the gap between increasingly powerful hardware capabilities and increasingly complex software development.
Calvin and Hobbes

Mike Peters

Everything familiar has disappeared! The world looks brand-new!

Wow, it really snowed last night! Isn't it wonderful?

A new year, a fresh, clean start!

It's like having a big white sheet of paper to draw on.

A day full of possibilities!

It's a magical world, Hobbes. Or, buddy...

Let's go exploring!
Appendix A

Previous publications

- *Parallel assertions for debugging parallel programs* [89]

  This paper introduced the parallel assertion statement. It included a discussion of sequential vs parallel assertions (Chapter 2), a sequentially consistent semantics for parallel assertions (Chapter 3), and a set of annotated PARSEC benchmarks (Chapter 4).

- **passert**: *A tool for debugging parallel programs* [88]

  This paper introduced passert, a practical runtime checker for parallel assertions. It described the implementation of the tool (Chapter 4), and summarized the results of the University of Michigan Collection of Concurrency Bugs study (Chapter 5).

- *Parallel Assertions for Architectures with Weak Memory Models* [90]

  This paper updated the semantics of parallel assertions to account for the weak memory models used in modern computers. It included a formal syntax for parallel assertions (Chapter 2), a weak-memory model semantics for parallel assertions (Chapter 3), and a proof of correctness of several optimizations given that semantics (Chapter 4).
Bibliography


