UNIVERSITY OF CALIFORNIA

SANTA CRUZ

DYNAMIC PREDICTION OF CONCURRENCY ERRORS

A dissertation submitted in partial satisfaction
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Caitlin Harrison Sadowski

June 2012

The Dissertation of
Caitlin Harrison Sadowski
is approved:

Professor E. James Whitehead, Jr.,
Chair

Professor Cormac Flanagan

Professor Scott A. Brandt

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Dynamic Prediction of Concurrency Errors

by

Caitlin Harrison Sadowski

Taking advantage of parallel processors often entails using concurrent software, where multiple threads work simultaneously. However, concurrent software suffers from a bevy of concurrency-specific errors such as data races and atomicity violations. Dynamic analysis, based on analyzing the sequence of operations (called a *trace*) from a source program as that program executes, is a powerful technique for finding concurrency errors in multithreaded programs. Unfortunately, dynamic analyses are confined to analyzing the observed trace.

Nonetheless, there are situations where a concurrency error does not manifest on a particular trace although it is intuitively clear that the program that produced that particular trace contains a concurrency error. The central research hypothesis explored by this dissertation is that dynamic analysis can discover concurrency errors that do not manifest on the observed trace.

This dissertation introduces a new relation, *causally-precedes* (CP), that enables precise predictive race detection, with no false positives. A single CP-based race detection run discovers several new races, unexposed by 10 independent runs of a traditional dynamic race detector. To further address dynamic predictive race detection, this dissertation introduces the *must-before* relation and accompanying dynamic analysis tool (EMBRACER) that is not precise but enables online prediction. Experimental results show that EMBRACER detects 20-23% more races than a traditional race detector alone for reactive programs.

This dissertation also introduces SIDETRACK, a lightweight dynamic atomicity analysis tool that generalizes from the observed trace to predict additional atomicity

violations. Experimental results show that this predictive ability increases the number of atomicity violations detected by SideTrack by 40%.

When developing these tools, it became clear that it was difficult to test them. For example, test programs that contain data races may be non-deterministic. A methodology for deterministic testing for dynamic analysis tools using trace snippets, described in this dissertation, alleviates this difficulty.

This work is dedicated

to Ian Pye and Jaeheon Yi,

for all their love and support.

Acknowledgments

Getting my PhD was a long process that depended on the encouragement, advice, and collaboration of many other people. I am especially grateful for my advisors, Jim Whitehead and Cormac Flanagan; I learned many invaluable research skills from both of them. I am indebted to my collaborators – Jaeheon Yi, Cormac Flanagan, Yannis Smaragdakis, and Jacob M. Evans – for allowing me to include our shared research in this dissertation. I received valuable assistance from the GAANN fellowship and the UCSC President's Dissertation Year Fellowship, as well as GSR funding from Scott Brandt and Wang-Chiew Tan. I thank my labmates – Jaeheon Yi, Kenn Knowles,Tom Austin and Tim Disney – for their moral and academic support throughout graduate school. I would also like to thank all of my other co-authors and collaborators: Andy Begel, Tom Ball, Sebastian Burckhardt, Daan Leijen, Ray Buse, Wes Weimer, Greg Levin, Ian Pye, Scott Brandt, Shelby Funk, Chris Lewis, Zhongpeng Lin, Xiaoyan Zhu, Sri Kurniawan, Andrew Shewmaker, Judith Bishop, Ganesh Gopalakrishnan, Joe Mayo, Shaz Qadeer, Madan Musuvathi, Steve Freund, Kenn Knowles, Alexandra Holloway, and Laurian Vega.

For my entire life, the unwavering support of my family has been behind me in everything I do. I would not be the person I am without it. My housemates, Amy Van Scoik, John Bitter, and (later) Tez Perez made it fun to come home early every night. My closest friends in graduate school – Greg Levin, Alexandra Holloway, and Rosie Wacha – made my time at school enjoyable.

Lastly, I would like to thank the two people I love the most in the entire world: Ian Pye and Jaeheon Yi. I would never have gone to graduate school without Ian, and I would never have finished this dissertation without Jaeheon. I am looking forward to beginning the next chapter of my life while knowing both of you.

Part I

Introduction

Chapter 1

Thesis

This dissertation is focused on pushing the state of the art of dynamic analyses targeted at concurrency errors. Specifically, the thesis is that dynamic analyses for concurrency errors can be made more powerful by using predictive techniques and relations. First, let us unpack some of these terms.

1.1 Concurrency

Once upon a time programmers were taught to write sequential programs, with the expectation that new hardware would make their programs perform faster. Around 2005, we hit a power wall [12]; the energy output of a chip with increased processor speed has become untenable. Today, all major chip manufacturers have switched to producing computers that contain more than one CPU [147]; parallel programming has rapidly moved from a special-purpose technique to standard practice in writing scalable programs. Taking advantage of parallel processors often entails using concurrent software, where multiple threads work simultaneously. However, concurrent software suffers from concurrency-specific errors [40, 93, 98], such as data races, atomicity violations, determinism violations, and deadlocks (defined in Section 1.3 and Chapter 4).

1.2 Dynamic Analysis

"Is my program correct?"

This is a central question when writing programs. What does it mean for a program to be correct? It is not immediately obvious without additional semantic information about the specific program. In order to check whether a program behaves as expected, you need to know both what the program does and a specification of what it should do.

The field of program verification and testing arises from the tension between an executable program and a high-level program specification. Program specifications come in many forms, ranging from informal comments or design documents, to executable unit tests or assertions, to lightweight type annotations, to machine checkable statements in a formal specification language.

Some specifications are general purpose, in that they apply to all programs. General purpose specifications include statements like "programs should not have a segmentation fault," "programs should not deadlock," or "programs should not have any buffer overflow errors." General purpose specifications are nice because they can be encoded at the program checker level instead of at the program level. General purpose specifications lead to general purpose program analysis systems; such systems are increasingly used commercially (*e.g.* Coverity [43], FindBugs [60], JLint [82], PMD [117], Klocwork [89]).

In general, checking if a program satisfies a specification is undecidable. *Precise* or *sound* analyses do not report false alarms; every error reported by such analyses represents a real error in the program. However, precise analyses achieve decidability because they may miss some errors. *Complete* analyses find all errors in a program; these analyses do not report false negatives, but may report false positives.

Static analysis involves analyzing the source code of a program; static type checking can be viewed as a lightweight static analysis system. Static analyses have potential for low precision, and so significant engineering effort must be devoted to pruning false alarm counts. We believe that false alarms significantly degrade the utility of analysis tools, implying that minimizing false alarms should be considered a key goal when developing an analysis. For this reason, we are focusing on *dynamic analysis*.

Dynamic analysis involves analyzing the sequence of operations from a source program as that program executes, also known as "online" or "on-the-fly" analysis (*e.g.* Valgrind [110]). This sequence is called a *trace*. As a general rule, dynamic analyses have an easier time minimizing false alarms, because of the presence of additional runtime information. However, dynamic analysis may miss some errors; for example, unexplored branches of conditionals are not visible at runtime. To overcome this limitation, dynamic analyses may be run multiple times on different traces for the same program.

In general, concurrency errors are particularly difficult to discover with testing – they typically manifest on rare thread schedules. Even after thousands or millions of test runs, there may be no observed anomalous behaviour. Luckily, many concurrency-specific errors violate general purpose specifications (*e.g.* "programs should not deadlock") that can be efficiently checked by dynamic analysis.

Dynamic analysis is a potent tool for finding insidious concurrency errors in a running program, without introducing too many false alarms, but is confined to analyzing the observed trace. This dissertation is focused on advancing the state of the art of dynamic analysis techniques for concurrency errors. In this dissertation, we specifically focus on two common classes of errors: data races and atomicity violations.

1.3 Data Races and Atomicity Violations

1.3.1 Data Race

When two accesses to the same shared variable occur by different threads and at least one of them is a write, we say the accesses are clashing.[1] A *data race* occurs when there are two *concurrent* clashing accesses, *i.e.* without any ordering provided by synchronization. The write and the other access are "racing" with each other, and the result has scheduler-dependent non-determinism. For example, the `BankAccount` class in Figure 1.1 does not have any synchronization: if two threads call the `deposit` method at about the same time (as in the trace in Figure 1.2), the results may be surprising.[2]

Figure 1.1: Example Class `BankAccount`

```
1  class BankAccount {
2    long amount;
3
4    void deposit(int d) {
5      long temp = amount;
6      amount = temp + d;
7    }
8  }
```

Figure 1.2: Race Condition for `BankAccount`

Thread 1	*Thread 2*
`r(amount)`	
	`r(amount)`
`w(amount)`	
	`w(amount)`

[1] In some prior papers (*e.g.* [62, 139]), these accesses were referred to as "conflicting." In this dissertation, we use the term "clashing" instead since there is a conflicting definition of "conflict" in the atomicity literature. See Section 7.3 of Chapter 7 for more information on clashes vs. conflicts.

[2] Our visual convention is that events occur top-to-bottom in the total order of the observed execution. We use the standard syntax `acq(l)`/`rel(l)` for the acquisition/release of lock `l`, and `w(x)`/`r(x)` for the write/read of variable `x`.

Data races are a common and costly source of errors in multithreaded programs. They have been implicated in region-wide electrical blackouts [70], problems during Mars Rover missions [102], and multiple deaths from radiation poisoning [94]. Although there has been some recent work focused on identification of "data race bugs" instead of all races [88], the idea that it is possible to have a benign race is hotly contested [4, 20].

Data races make concurrent software unpredictable and problematic because the behaviour of a racy program reverts to being machine-dependent; racy programs necessitate reasoning about low-level details such as the granularity of memory [21]. In contrast, race-free programs have predictable and intuitive behaviour; they are easier to understand and debug.

1.3.2 Atomicity Violations

Race-free programs may still exhibit unintended thread interference, because race freedom is a low-level property dealing with memory accesses. That is, higher-level semantic notions of thread non-interference are not addressed by race freedom.

Atomicity is a semantic non-interference property based on grouping blocks of code together for easier reasoning. Informally, a block of code is atomic if, for all executions of that code block (known as *transactions*), the effect of that execution can be considered in isolation from the rest of the running program. In the database literature this property is often called *serializability* [17]. In a serial trace, all transactions execute without any interleaved operations by other threads. For example, accesses to the `deposit` method in the trace in Figure 1.4 for the (corrected) `BankAccount` class from Figure 1.3 are serial and hence atomic. In a serializable trace, although operations may be interleaved with transactions, the trace is *equivalent* to (*e.g.* has the same behaviour as) a serial trace. In other words, atomicity guarantees that a program's behaviour can be understood as if each atomic block executes without interference from other threads.

Figure 1.3: Example Program `BankAccount` with Correct Synchronization

```
1  class BankAccount {
2    long amount;
3    Object mutex;
4
5    void deposit(int d) {
6      synchronized(mutex) {
7        long temp = amount;
8        amount = temp + d;
9      }
10   }
11 }
```

Figure 1.4: Trace for Corrected `BankAccount`

Thread 1	*Thread 2*
`acq(mutex)`	
`r(amount)`	
`w(amount)`	
`rel(mutex)`	
	`acq(mutex)`
	`r(amount)`
	`w(amount)`
	`rel(mutex)`

Atomicity coarsens the necessary reasoning about potential interleavings between threads to the level of atomic blocks instead of individual operations. Inside an atomic block, simpler sequential reasoning may be applied to show correctness. Atomicity is used informally to organize and reason about multithreaded code; for example, most Java methods are atomic [61].

An atomicity violation occurs when an outside thread "interferes" with the operations inside an atomic block. For example, reading the same variable twice inside an atomic block and getting different values because of an external update indicates an atomicity violation. Even though each individual read may be protected by a lock, both reads together should form part of a larger atomic action. A *synchronized block* is a

Figure 1.5: Example Program `BankAccount` with Synchronization

```
1  class BankAccount {
2    long amount;
3    Object mutex;
4
5    void deposit(int d) {
6      synchronized(mutex) {
7        long temp = amount;
8      }
9      synchronized (mutex) {
10       amount = temp + d;
11     }
12   }
13 }
```

Figure 1.6: Atomicity Violation for `BankAccount`

Thread 1	*Thread 2*
`acq(mutex)`	
`r(amount)`	
`rel(mutex)`	
	`acq(mutex)`
	`r(amount)`
	`rel(mutex)`
`acq(mutex)`	
`w(amount)`	
`rel(mutex)`	
	`acq(mutex)`
	`w(amount)`
	`rel(mutex)`

lock acquire, followed by a potentially empty sequence of operations by the same thread, followed by a lock release. In Figure 1.5, although we have added synchronization to the `BankAccount` class, thus removing the race on `amount`, the synchronized blocks are not large enough to make the `deposit` method atomic. For example, the possible trace in Figure 1.6 has many of the same unexpected results as in Figure 1.2. In Java, writes to `long`s theoretically have the potential to be nonatomic,[3] and so the synchronization

[3]In practice, writes do behave atomically on most modern hardware.

in Figure 1.5 does provide some benefit. Although atomicity appears to necessitate annotations that specify the atomic blocks, it is also possible to check for atomicity violations at the level of all methods [61, 66] or all synchronized blocks [115].

1.4 Problem Space

In the previous discussion in Section 1.3.1, a data race is defined as two *concurrent* accesses to the same shared variable, where at least one of them is a write. It can be difficult to tell whether two accesses could be concurrent, so previous race detectors (described in Section 6.2.1) and atomicity analyses (described in Section 6.3) have relied on a partial order called the happens-before (HB) relation to give an under-approximation of when accesses are concurrent. This relation, originally developed by Lamport [91], identifies dependencies within a program trace. The happens-before relation gives us a notion of trace equivalence, in that traces with the same happens-before order exhibit the same behaviour.

Specifically, happens-before analyses attempt to discern when there has been inter-thread communication that effectively orders two clashing accesses. Unordered accesses are reported as a race, since there is no reason why they could not have occurred at exactly the same instance. In its simplest form, happens-before is a partial order that captures the logical ordering between operations in a multithreaded program's execution by:

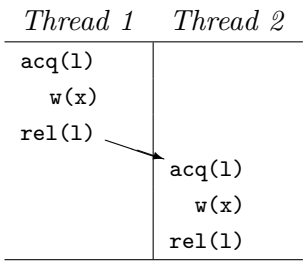
- ordering all events by a single thread in the order they were actually observed
- ordering lock releases and subsequent acquisitions of the same lock in the order they were observed

Under the assumption that threads can only communicate via mechanisms represented in the happens-before order, this approach is precise. Although the simplistic defini-

tion presented above only covers locks, other inter-thread communication (*e.g.* forking and joining of threads or accesses to volatile variables) can be captured as happens-before edges. The happens-before relation is discussed in more detail in Section 7.3 of Chapter 7.

As an example of an execution trace with no race, consider Figure 1.7. As before, our visual convention is that events occur top-to-bottom in the total order of the observed execution and we use the standard syntax `acq(l)`/`rel(l)` for the acquisition/release of lock `l`, and `w(x)`/`r(x)` for the write/read of variable `x`. In this trace, all writes to `x` are protected by lock `l` and occur inside a synchronized block. The happens-before relation contains an edge between the release of `l` by Thread 1 and the acquire of `l` by Thread 2 (represented in the figure with an arrow). This edge reflects the presence of a dependency between the two writes to `x` such that they cannot occur at the same time.

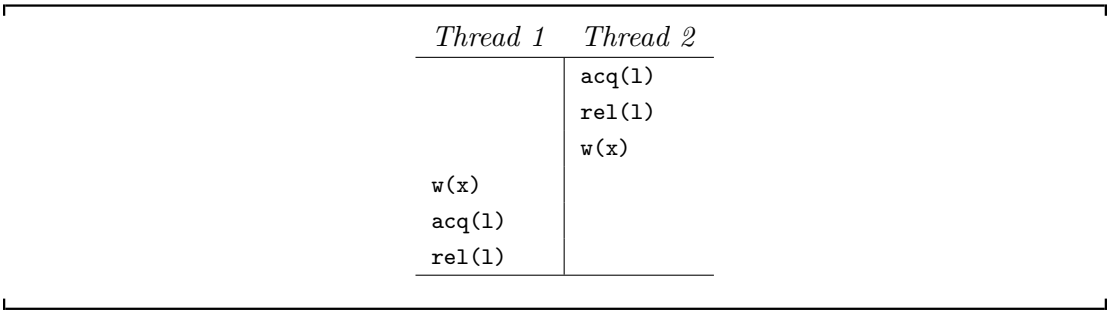
Figure 1.7: Example where all access to `x` are protected by lock `l`.



As an example of a racy execution trace, consider Figure 1.8. In this trace, the two writes to `x` do happen at essentially the same time. In fact, the happens-before relation contains no dependencies between the two writes.

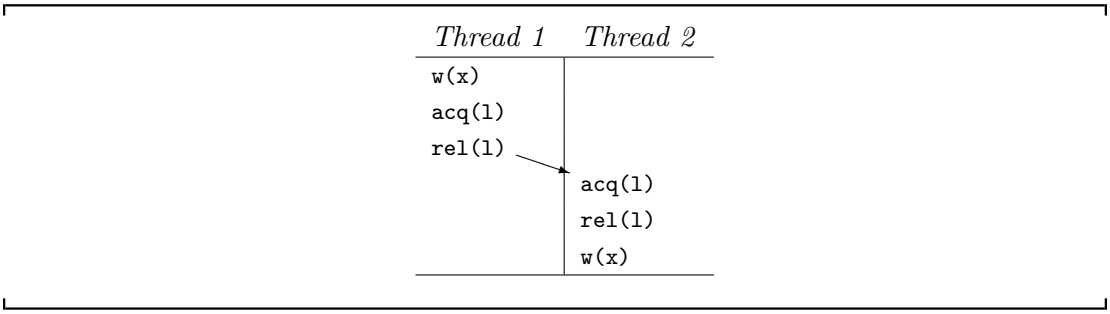
However, the same program that produced the trace in Figure 1.8 could have easily produced the trace in Figure 1.9. In this trace, the writes to `x` are ordered under the happens-before relation via an edge from the release of `l` by Thread 1 to the subsequent acquire by Thread 2. When considering the trace in Figure 1.9, it is intuitively

Figure 1.8: Example with a very simple happens-before race on `x`.



clear that the program that produced this trace contains a race. Unfortunately, the happens-before relation is overly restrictive in this case.

Figure 1.9: Example with a certain race on `x`, but no happens-before race.



The example from Figure 1.9 is at the crux of this dissertation. Even with only the limited information of the dynamic trace produced from one run of a program, there are situations where we can *predict* that another execution exists (as in Figure 1.8) that contains a happens-before race.

This predictive intuition also holds for detecting atomicity violations. Recall that a trace is serializable if it is equivalent to a serial trace. Previous dynamic atomicity analyses use a cycle-based algorithm with transactions as nodes and report an error if and only if the observed trace is not serializable [66]. Essentially, these algorithms build transactional happens-before graphs to detect cycles, which correspond directly to atomicity violations.

To illustrate prediction as applied to atomicity violations, consider the trace in

Figure 1.10. Here, `beg(a)` and `end(a)` demarcate the begin and end of an atomic block (labelled `a`) within the execution trace. In this trace, Thread 1 is executing an atomic block containing two synchronized blocks and Thread 2 is executing a single synchronized block. Each synchronized block contains an unspecified (possibly empty) sequence of operations denoted by "...". As before, the vertical ordering of the statements of the two threads reflects their relative execution order.

Figure 1.10: Observed Serial (Hence Serializable) Trace

Thread 1	*Thread 2*
`beg(a)`	
`acq(l)`	
`...`	
`rel(l)`	
`acq(l)`	
`...`	
`rel(l)`	
`end(a)`	
	`acq(l)`
	`...`
	`rel(l)`

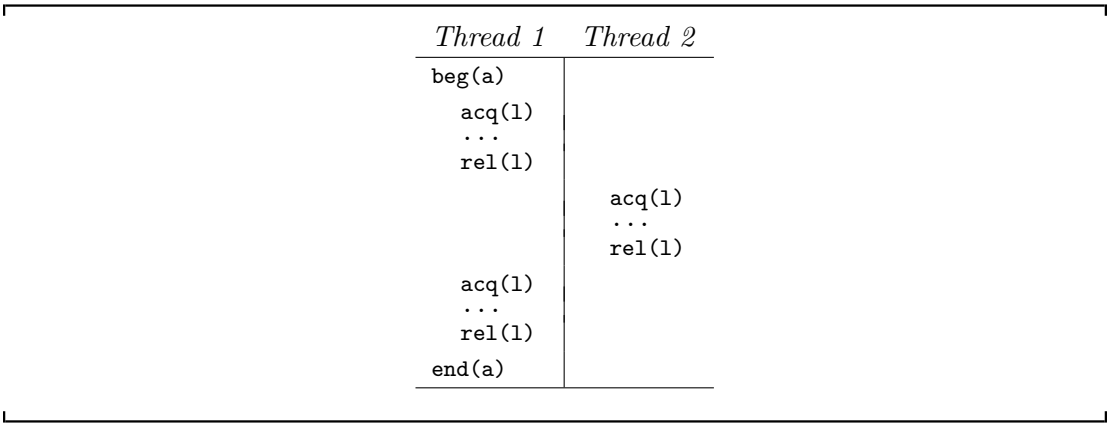
Clearly, this trace is serial and hence trivially serializable. A careful analysis of this trace, however, shows that the synchronized statement of Thread 2 *could have been scheduled* in between the two synchronized blocks of Thread 1; the original source program that generated the trace of Figure 1.10 is also capable of generating the non-serializable trace shown in Figure 1.11.

1.5 Research Question

Our central research hypothesis is:

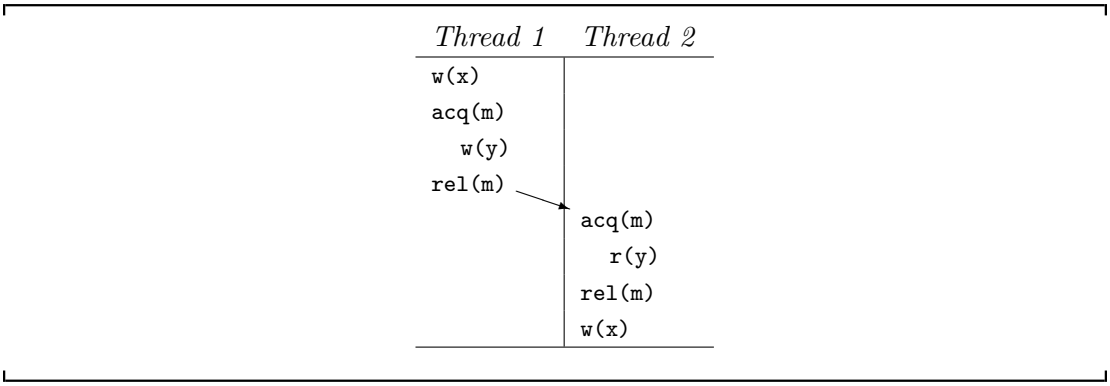
> *Dynamic analysis can discover concurrency errors that do* not *manifest on the observed trace.*

Figure 1.11: Feasible Non-Serializable Trace



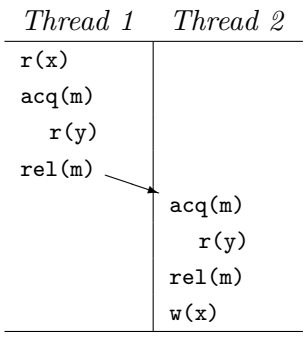
In addition to the trace observed on one run of a program, many other *feasible* traces exist that could be generated by a given program. The key idea explored by this dissertation is identifying situations, like those described previously, where two synchronized blocks could be reordered in an observed trace, even though they synchronize on the same lock. Previous related online analyses (described in Chapter II) do not consider reorderings of synchronized blocks on the same lock.

Figure 1.12: A trace where reordering the synchronized block on `m` could lead to divergent behaviour.



It is challenging to identify situations where synchronized blocks can be reordered. For example, consider the trace in Figure 1.12. Despite the fact that the accesses to `x` are not directly protected by any synchronization, it is not clear whether the two writes to `x` can happen at the same time. If the read of `y` by Thread 2 were

Figure 1.13: Example with a certain race, but no happens-before race.



to occur prior to the write of y by Thread 1, it could have read a different value. By reading a different value, the subsequent operations by Thread 2 may be completely different, *e.g.* if the read of y is used in the guard of a conditional. In particular, it is possible that every trace in which the read of y by Thread 2 occurs prior to the write of y by Thread 1 is race free. In contrast, the synchronized blocks containing the reads of y in the trace in Figure 1.13 can be reordered without altering the program behaviour. If we are able to predict, given an observed trace, that a race condition occurs on another feasible trace, we say there is a *predictable* race. More diabolical examples of traces that are not guaranteed to have predictable races are discussed in Part 12.

In this dissertation, we present causally-precedes, a weaker relation than the happens-before relation, yet offering the same desirable features: CP leads to precise race detection, and can be evaluated efficiently (in polynomial time). Multiple researchers have fruitlessly pursued such a weakening of HB in the past [139]. We demonstrate with numerous examples why it is not easy to weaken HB without introducing false positives. A single CP-based race detection run discovers several new races, unexposed by 10 independent runs of plain HB race detection.

We then develop a *must-before* relation that is not precise but enables online prediction. We formalize a notion of *must-before races* that provides better coverage than happens-before races. We develop a dynamic race prediction algorithm called

EMBRACER for must-before races. We present experimental results showing that EMBRACER detects 20-23% more races than a happens-before detector alone for reactive programs.

We show how serializable traces can still reveal atomicity violations in the original program and present SIDETRACK, a lightweight dynamic analysis tool that generalizes from the observed trace to predict atomicity violations that could occur on other feasible traces. We present experimental results showing that this predictive ability increases the number of atomicity violations detected by SIDETRACK by 40%. Despite performing online prediction, SIDETRACK offers performance competitive with related dynamic atomicity analyses.

When developing SIDETRACK and EMBRACER, we discovered that it was difficult to test them. For example, test programs that contain data races may be nondeterministic. To alleviate this difficulty, we present a methodology for deterministic testing for dynamic analysis tools using trace snippets. This methodology involves a language-agnostic domain-specific language (DSL) for describing trace behaviour, Tiddle. To use Tiddle, we present a compiler, implemented in Haskell, that translates Tiddle traces into deterministic concurrent Java programs.

Chapter 2

Organization

Before we become enmeshed in the details, let us describe the arc of this dissertation. In the next two parts (Part II and Part III), we provide a foundation for understanding and contextualizing the remainder of this dissertation. Following this foundation, we present algorithms for predictive data race detection (Part IV and Part V) and predictive atomicity violation detection (Part VI). We also present a language for testing dynamic analysis tools (Part VII).

Part II comprises Chapters 3–6. This part is focused on contextualizing this dissertation with other related work. We first describe some as yet implicit assumptions about the concurrency and memory model used (Chapter 3). We then define some common concurrency errors, to enable the description of analyses for detecting such errors (Chapter 4). Following these foundations and definitions, Chapter 5 is focused on methods for testing concurrent programs and program analyses. Finally, Chapter 6 contains an overview of other analysis tools for detecting the concurrency errors identified.

After describing this context, we review a selection of technical background (Part III, comprised of Chapters 7–9). We clarify semantics and terminology built upon in subsequent chapters (Chapter 7 and 8) and describe the analysis framework used to implement the tools presented in Part V and VI (Chapter 9).

Part IV, comprised of Chapters 10–12, is focused on the causally-precedes (CP) relation: a novel relation over the operations in a trace that is less restrictive than the happens-before relation and so identifies more concurrent accesses.[1] A precise race detector can be built from this relation with polynomial complexity that does not introduce false positives. Chapter 11 introduces the CP relation. The subtleties of CP are illustrated through examples of diabolical traces in Chapter 12.

The CP relation is a major theoretical contribution to the field of dynamic race detection. However, this relation is not easy to translate into a dynamic analysis algorithm. Part V, comprised of Chapters 13–17, is focused on a novel dynamic analysis algorithm for race prediction.[2] Even when the observed trace is race-free, this algorithm predicts if a race may occur on another, similar trace from the same program. This algorithm is not guaranteed to be sound, but enables online prediction. We present the predictive must-before (MB) relation in Chapter 14, and a race detection algorithm built from this relation (Embracer) in Chapter 15. We discuss the implementation of Embracer in Chapter 16 and evaluate this implementation in Chapter 17.

In Part VI, comprised of Chapters 18–22, we focus on dynamically detecting atomicity violations in traditional multithreaded programs.[3] Even if the observed trace is serializable, our online analysis can still infer that the original program can generate other feasible traces that are not serializable. We present the SideTrack analysis (Chapter 20) along with a practical implementation (Chapter 21) and evaluate this implementation (Chapter 22).

Part VII, comprised of Chapters 23–24 is focused on Tiddle: a language-agnostic domain-specific language (DSL) for describing trace behaviour. Tiddle can

[1] This work was originally presented at the 2012 Symposium on Principles of Programming Languages (POPL) [139] and was developed jointly with Yannis Smaragdakis, Jacob Evans, Jaeheon Yi, and Cormac Flanagan.

[2] This work was developed jointly with Jaeheon Yi and Cormac Flanagan.

[3] This work was originally presented at the 2009 Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD) [171] and was developed jointly with Jaeheon Yi and Cormac Flanagan.

be used to generate deterministic tests for validating dynamic analysis tools.[4] Chapter 23 introduces Tiddle and highlights the use cases for this language. Chapter 24 describes the Tiddle grammar and compiler.

Finally, in Part VIII (consisting of Chapter 25), we highlight some areas for future work. This dissertation also includes two appendices of material that is reproduced here for convenience: Appendix A contains experimental results for CP-based race detection and Appendix B contains the soundness proof for the CP relation [139]. Part X contains the bibliography.

[4]This work was originally presented at the 2009 International Workshop on Dynamic Analysis (WODA) [127] and was developed jointly with Jaeheon Yi.

Part II

Related Work

Chapter 3

Concurrency and Memory Models

3.1 Concurrency Model

At a low level, multiprocessor systems often involve explicit multiple threads of control that can access the same data (*shared memory*). Multithreading with shared memory is the most popular of three dominant concurrency models at the language level, and is the concurrency model for the analyses described in this dissertation. This model, though popular, is very difficult to reason about [93] Programs can also be structured to map operations onto disjoint data (*data parallelism*) or use messages to communicate without sharing memory directly (*message passing*). Many of the concurrency errors discussed in subsequent sections could be prevented by adoption of a concurrency model that does not involve shared memory;[1] further discussion of alternate concurrency models is beyond the scope of this dissertation. Since multithreading and shared memory are components of mainstream languages like Java and C#, this concurrency model is very widely used and will be a feature of legacy code for the foreseeable future.

[1]These other models also have limitations. Data parallelism works well for divide-and-conquer type problems, and less well for other problem structures. Message passing programs may contain timing bugs; for example, in a language with asynchronous message passing, one thread may wait forever for a message that was previously sent by another thread.

3.2 Memory Models

A *trace* describes the operations performed by an executing program. For single-threaded programs, a trace is inherently sequential and forms a total order on program operations. For multithreaded programs, a trace is actually a *partial* order over program operations; operations by different threads may happen at essentially the same time. However, the dynamic analyses presented in the dissertation take as input a representative total order over the program operations by all threads.

A memory model defines the values that may be obtained by a read operation in a trace. A memory model can be viewed as a contract between the programmer, compiler, and hardware about what sorts of reorderings and optimizations are acceptable. *Sequential consistency* is the most intuitive memory model in that every read operation will observe the value stored by the last write. A *relaxed* memory model is one where the observational ordering constraint is relaxed to facilitate certain performance optimizations, such as those enacted by the compiler and hardware. The Java Memory Model [99] is an example of a relaxed memory model.

In Java, programs without happens-before data races follow sequential consistency. The tools presented in Parts IV and V will report a happens-before race if one exists. If there are no happens-before races in a trace, these tools can leverage sequential consistency as an assumption when predicting races for that trace.

Chapter 4

Concurrency Errors

In Section 1.3 of Chapter 1 we defined race conditions and atomicity violations. The mere presence of data races is enough to break the guarantee of sequential consistency on most hardware platforms. While atomicity is related to the notion of race-freedom [111], these two correctness properties provide complementary guarantees. Race-freedom guarantees that the program behaves as if executed on a sequentially-consistent memory model [5], while atomicity guarantees that each atomic block behaves as if executed serially.

This dissertation is focused on atomicity violation and data race detection, although there are other potential concurrency errors in multithreaded programs. In this chapter we outline three additional concurrency-specific errors which programmers need to contend with. All of these errors cause concurrency bugs, and programmers need to reason about these types of errors to ensure their code is correct.

4.1 Cooperability Violations

One major problem with atomicity as a central correctness property is that atomicity only gives you guarantees about code that is inside atomic blocks. Code

that is outside of atomic blocks is subject to *preemptive* reasoning, in which a context switch may occur at any program point. In other words, atomicity enforces a bimodal sequential/preemptive reasoning; code inside of atomic blocks has simpler sequential reasoning than code outside of an atomic block.

Instead of specifying atomic code sections where interference should *not* occur, *cooperative semantics* permits context switches only at explicitly marked yield annotations. Cooperative semantics allows for mostly intuitive, sequential reasoning, except at specific yield annotations.

Since only allowing context switches at yield points would be excessively restrictive, the correctness criterion of *cooperability* [170] is that code executes *as if* context switches happen only at specific yield points. Thus, even though cooperable programs execute under preemptive semantics, we can reason about their correctness *as if* they execute under a cooperative semantics.

The example in Figure 4.1 illustrates these concepts. The `Ticket` class has a buggy `valuate()` method that is unsafe under different thread interleavings. The two yield annotations in the program represents all observable interference, and so the program is cooperable. The two traces represent program execution under preemptive and cooperative scheduling; their behavior is equivalent. The property of cooperability allows us to reason about the buggy behavior of trace (a) using the cooperative trace (b).

4.2 Determinism Violations

Determinism is a key property when dealing with concurrency. The importance of determinism is understandable; non-determinism is what makes other types of concurrency errors both problematic and difficult to detect. For deterministic programs, which always behave the same given a particular input, it is not necessary to

Figure 4.1: A cooperable program with two equivalent traces.

```
1  class Ticket {
2    float value;
3    public void valuate() {
4      float tmp1 = value * 0.10;
5      /* yield; */
6      float tmp2 = value + tmp1;
7      /* yield; */
8      value = tmp2;
9    }
10 }
```

Thread 1	*Thread 2*
rd value	
	rd value
wr tmp1	
	wr tmp1
yield	
	yield
rd tmp1	
	rd tmp1
rd value	
	rd value
wr tmp2	
	wr tmp2
yield	
	yield
	rd tmp2
rd tmp2	
	wr value
wr value	

(a) A preemptive scheduling

Thread 1	*Thread 2*
rd value	
wr tmp1	
yield	
	rd value
	wr tmp1
	yield
rd tmp1	
rd value	
wr tmp2	
yield	
	rd tmp1
	rd value
	wr tmp2
	yield
	rd tmp2
	wr value
rd tmp2	
wr value	

(b) An equivalent cooperative scheduling

reason about particular thread schedules. For example, if the result of a data race is deterministic (*e.g.* two threads writing the same value to a variable), then the data race itself is benign; the result of the race does not matter.

Even if an entire program is not deterministic, subcomputations may be. Atomicity violations are problematic because they break the determinism of the operations making up an atomic block.

4.3 Deadlocks

Many solutions to fix concurrency errors like atomicity violations or data races involve adding additional synchronization to a program. Care needs to be taken with the introduction of synchronization, or deadlock can occur. Deadlocks result from attempted cyclic lock allocation between at least two threads. For example, if one thread attempts to acquire lock A followed by lock B whilst another thread attempts to acquire lock B followed by lock A, both threads may block indefinitely waiting for a lock to be released.

Deadlocks, like other concurrency errors, may not manifest under testing. However, whereas data races, atomicity violations, cooperability violations, and determinism violations can cause insidious anomalous behaviour that may or may not crash the executing program, the manifestation of a deadlock is typically more obvious. For example, if all threads are deadlocked, CPU usage drops down to 0% and the program stops in its tracks. Deadlock detection and repair tends to be more straightforward than race detection.

Chapter 5

Testing and Concurrency

Before we present work related to program analysis for detecting concurrency errors, we first review a selection of the work related to concurrency testing in specific. All of the work presented in this chapter is orthogonal to the program analysis techniques introduced by this dissertation and described in Chapter 6. Dynamic program analysis typically relies on tests to run the program (and produce a trace). Also, the techniques described in this chapter are often combined with program analysis techniques (such as those described in Chapter 6), so as to identify specific classes of concurrency errors whilst testing programs. For example, a developer may want to verify that there are no data races while running a test.

There have been two main directions for testing concurrent programs. Some systems attempt to explore the space of possible schedules to find more bugs. These can easily be combined with program analysis tools, for example to increase coverage of dynamic analyses. Other systems focus on controlling nondeterminism in tests, either through scheduling annotations or via deterministic replay. Some of these techniques also extend to testing program analyses.[1]

[1] In Part VII we will elaborate on Tiddle: a language for describing program execution traces which serve as test inputs for dynamic analyses.

5.1 Scheduler-Driven Approaches

Dynamic analyses run over a particular execution trace, and so can be combined with a system which explores multiple schedules to cover a larger variety of program traces. Systems which explore the space of possible schedules fall into three main groups.

Model Checking Direct execution model checking involves repeatedly running a program until all possible interleavings have been explored; this approach would find all errors on all feasible traces if run to completion. This approach suffers from scalability concerns due to an explosion of potential state spaces to explore. Java PathFinder [158] is a stateful direct execution model checker implemented as a Java virtual machine. CHESS [105, 106, 107] is a stateless direct execution model checker that uses iterative context bounding (*i.e.* prioritizing schedules with fewer context switches) as an exploration heuristic and also as an effective coverage metric. Gambit [42] leverages a combination of heuristic-based exploration followed by systematic checking to provide both fast likely bug detection and good coverage. ALPACA [125] unit tests check for concurrency-specific errors (such as race conditions or deadlocks), control scheduler-dependent non-determinism, and run the unit tests on different thread schedules. ALPACA unit tests are built off of CHESS. Other researchers have also explored the combination of unit tests (JUnit) and model checking (JPF) [148], although ALPACA is the most robust system to do so.

Fuzzing The second strategy primarily uses heuristics to drive exploration to interesting schedules. *Noise-makers* insert `sleep()` or `yield()` statements at heuristically-chosen points [49]. Fuzzing [134, 26] involves using a combination of randomized scheduling and directed scheduling to attempt to schedule concurrency errors like data races and atomicity violations. Though fuzzing does not have the scalability concerns of

model checking, it is nondeterministic and not guaranteed to find all possible errors.

Stress Testing In contrast, stress testing systems make no attempt to control the scheduler directly. Automatic or semi-automatic systems make it easy to run multiple copies of a test simultaneously on different threads [75]. A tool for unit testing concurrent business code [122] provides support for specifying which methods should be run concurrently, without requiring manual management of threading. Ricken *et al.* [123] present ConcJUnit: a framework built off of JUnit [85] that ensures that all threads spawned in unit tests terminate and that failing assertions or exceptions in any thread cause a test to fail. This framework leverages JUnit support for running multiple instances of tests. Since stress testing techniques, such as ConcJUnit tests, do not control scheduling, they may be nondeterministic and are not guaranteed to report all errors.

5.2 Deterministic Tests

Deterministic testing could be used to ensure that the results from a dynamic analysis run are repeatable. Unfortunately, deterministic testing strategies tend to have a high overhead. Two popular strategies exist for reproducible, deterministic, testing. Replay systems focus on re-execution of observed traces. Annotation-based systems allow programmers to explicitly specify desired schedules. An alternative strategy is to validate the determinism of a test (see Section 6.5).

Annotation-based Annotation-based systems are used for writing small (*e.g.* unit) tests. For larger tests the scheduling is too complex to effectively annotate.

MultithreadedTC [121] is a framework that allows programmers to encode scheduling information inside of unit tests. An external clock is added to the tests, along with a thread responsible for clock maintenance and deadlock detection. Test snippets can block waiting for clock ticks or assert when a tick has happened; ticks

occur when all threads in the system except the maintenance thread are blocked. The trace of operations is not made explicit in this system, and so it is possible to accidentally generate nondeterministic tests. MultithreadedTC was inspired by ConAn [97], a script based testing framework that also uses a clock ticks to enforce specific interleavings. Other researchers have proposed support for explicit specification of sets of schedules [79].

IMUnit [78] introduces a framework which allows developers to name particular code points (events) and define specific schedules involving those points, using Java annotations. Developers can also refer to named threads in these schedules. This framework can be used to control scheduling in unit tests, or to verify whether a specific execution matches a schedule. The authors also developed an Eclipse plugin which can be used to migrate sleep-based unit tests to the new system.

Deterministic Replay There is variety of prior research on *deterministic replay* of multithreaded applications [29, 37, 124, 133].[2] Essentially, these systems record information about thread scheduling so that a buggy execution trace can be replayed after a crash occurs. Without a replay system in place, it may be difficult to reproduce failures; different interleavings may not result in an error state. ConTest [49] combines replay with noise-making in one framework. One problem with replay-based methods is that once code has been modified, the logged schedule may no longer be valid. If this schedule is not transparent to programmers, it may be difficult to understand why a test can no longer be replayed. Another issue is that replay systems rely on a log of program events. This log is typically expensive (both in terms of time and space) to collect and to use during reply.

[2]Other replay systems exist for Java [84, 146], but here we are focused on replay systems targeted at concurrency.

5.3 Describing Program Traces

There has been some previous work on describing program traces for various purposes. A formal look at trace models of Prolog programs is explored in [80] for building high-level Prolog debuggers. Several authors have looked at simplifying, organizing and abstracting stored traces [25, 31]. The Test Behaviour Language (TBL) for traces provides a concise way of describing and testing for trace properties [31]. Various temporal logics can express bug patterns [22] or other arbitrary properties [44], and runtime verification systems [33] may monitor traces for violations of patterns and properties.

Some previous work uses traces to describe patterns. Generated traces that exhibit a statistical profile are used to test architecture performance characteristics [50]. Traces are mined for patterns of procedure calls [72].

5.4 Concurrency Benchmarks for Research

There has been a recent drive to create a benchmark of concurrency bugs, targeted at dynamic analyses [54, 55]. Eytani *et al.* claim that their benchmark has had an impact on the research community, and is now used by several groups. The concurrency bugs in the benchmark follow different bug patterns, but were mostly found in programs written by novices.

In TMUnit [73], a simple description language is used to generate workloads and test the semantics of transactional memories. The user specifies threads, transactions and their schedules to test a particular interleaving on a Software Transactional Memory (STM) system. The user can also specify invariants to be checked at each stage of the run.[3] Harmanci *et al.* validate their work by testing five STM systems. This work highlights the relevance of a trace DSL beyond dynamic analysis frameworks.

[3] In contrast, specifications in Tiddle (Part VII) of whether or not the dynamic analysis flags an error exist at a level above the Tiddle program itself.

Chapter 6

Detecting Concurrency Errors

As outlined in Chapter 1, developers use a variety of general purpose specifications, such as "programs should not deadlock," when reasoning about their programs. General purpose specifications are amenable to program analysis. This chapter overviews a variety of analysis tools for concurrency-focused general-purpose specifications. Specifically, we review analysis tools that detect the concurrency errors presented in Chapter 4.

6.1 Predictive Approaches

There is selection of prior theoretical work related to predictive and generalizing concurrency analysis [52, 59, 86, 138, 90, 87, 142, 163, 164, 166]. However, most of this work suffers from problems with precision and scalability. This work also typically involves incorporating the results of static analyses.

The most relevant prior concurrency error prediction work to this dissertation is based on a concept called *sliced causality* [34, 35, 36, 129, 136]. Sliced causality makes use of *a priori* control- and data-flow dependence information to obtain a reduced "slice" of a trace targeted towards a particular variable. For example, a race condition

on x is predicted with sliced causality by logging only operations relevant to accessing x in a program slice and model-checking all feasible slice permutations in a post-mortem analysis phase. These permutations are feasible because one may infer and ignore irrelevant operations via static dependence information; the precision of the predictions follow from the feasibility of the trace permutations generated. Sliced causality requires a separate trace prediction phase for each observed variable, is limited by the length of the logged slice, and critically depends on a whole-program static analysis to preserve precision of predictions.

We further discuss the most related predictive approaches for data race detection in Section 6.2.4 and atomicity violation detection in Section 6.3.1.

6.2 Race Detection

Several *static* approaches exist to deal with data races. Type systems [2, 24, 69, 130] and languages [14] have been proposed to prevent races in programs. Other static approaches include Warlock [145] and Locksmith [120] for ANSI C programs, scalable whole-program analyses [109, 162] and dataflow-analysis-based approaches [48, 53]. Aiken and Gay [9] also investigate static race detection focusing on SPMD programs. As mentioned previously, all of these approaches suffer from many false positives.

Post-mortem techniques [6, 39, 124, 149] for detecting races involve dynamic logging of operations that are then analyzed offline. These techniques may have scalability problems for programs that run for a long time. In particular, the logged trace size (and hence time to analyze the trace in the offline phase) may grow very large.

Some initial dynamic analyses for race detection were proposed in [46, 132]; today there are three main approaches. Precise detectors are based on *happens-before* analysis of the program trace [41, 62, 104, 118, 132]. Efficient but imprecise detectors are based on checking whether variables are consistently protected by the same locks [8,

38, 112, 131, 160]. *Hybrid* approaches combine happens-before analysis and locksets [51, 113, 119, 173].

6.2.1 Happens-Before

Numerous race detection tools are based on Lamport's happens-before (HB) relation [15, 41, 62, 104, 118, 132].[1] These tools are more precise than lockset-based race detectors (discussed in the following subsection) but are often less efficient.

The most efficient happens-before race detectors use vector clocks [103] to compactly track the happens-before relation on operations within a trace. A vector clock gives a relative impression of program timing by mapping thread identifiers to integer clocks. If k is the vector clock for an operation a in a trace, then $k(t)$ identifies the operations of thread t that happen-before a. Vector clocks are discussed in more detail in Chapter 8.

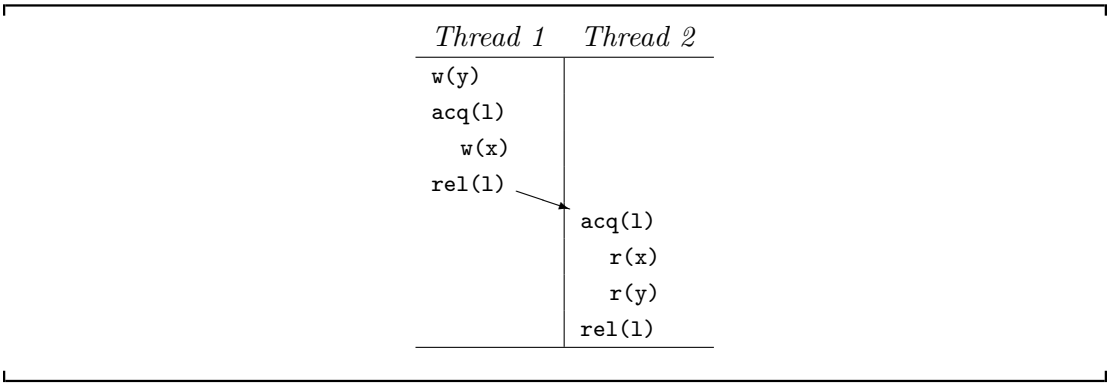
TRaDE [41] uses accordion clocks, vector clocks that shrink when all objects referenced by dead threads have been garbage collected, along with dynamic escape analysis (which determines the scope of pointers), to boost performance. The DJIT$^+$ [118] algorithm is an efficient happens-before vector clock algorithm that obviates the need for expensive full vector clock comparisons in special situations (*e.g.* when a thread reads the same variable twice in a row) for a 2-3x performance improvement. FastTrack [62] improves upon DJIT$^+$by using an adaptive lightweight representation for the happens-before relation and by introducing optimized constant-time fast paths that account for approximately 96% of operations encountered in a trace, and provides a 2.3x performance improvement over DJIT$^+$. This performance gain puts FastTrack on par with lockset-based tools.

[1]The happens-before relation is briefly described in Section 1.4 of Chapter 1 and is discussed in more detail in Section 7.3 of Chapter 7.

6.2.2 Lockset Approaches

A *lockset* for a shared variable is the set of locks that consistently protect access to that variable. Locksets were introduced [47] as an alternative representation to the happens-before analysis. Later, locksets were used alone as an efficient technique for data race detection in Eraser [131]. Unfortunately, programmers use a variety of different synchronization idioms to protect shared variables. Eraser gets many false positives caused by variables which are not consistently protected by the same lockset, yet are race free (*e.g.* using semaphore-based synchronization).

Figure 6.1: Example of no race on variable `y`.



Consider, for instance, the example execution in Figure 6.1. Although the two accesses to shared variable `y` do not occur with the same lock held, they are well-ordered—the second access only occurs after Thread 2 has observed a value written by Thread 1. It is quite possible that, if the two critical sections over lock `l` had been swapped, Thread 2 would not have attempted to read `y` since its read of `x` would have yielded a different value. In contrast, the inter-thread happens-before edge (shown as an arrow), together with the transitivity of the happens-before partial order, ensure that no happens-before race is reported by ordering the accesses to `y`.

Lockset-based approaches tend to be efficient, but are susceptible to high false alarm rates. Moreover, they are not well-suited for providing specific traces that lead

to the occurrence of a race condition.

Various refinements and extensions for Eraser have been proposed. Static escape analysis can improve performance [112, 160]. Reasoning about races at the object level instead of the memory word level [160] improves performance but leads to more false alarms. The lockset technique was also extended with timing *thread segments* [74] to reduce false positives caused by data not being protected by a lock during an initialization phase. Further performance enhancements use whole-program static analysis to reduce the amount of instrumentation necessary [38] or involve type inference [8]. Eraser's algorithm has also been implemented with AspectJ [19].

6.2.3 Hybrid Techniques

Recent work on dynamic data race detection has focused on combining locksets and happens-before analysis. These approaches leverage locksets for increased efficiency, but may introduce false positives. O'Callahan and Choi [113] developed a two-phase localization scheme; a first pass lockset analysis filters out problematic fields for a second pass hybrid analysis. RaceTrack [173] uses a happens-before analysis to estimate whether threads can concurrently access a memory location so as to reduce false positives caused by empty locksets. MultiRace [119] presents improved versions of happens-before and lockset algorithms. Locksets can also enable happens-before approaches to report additional warnings and reduce the number of vector clock comparisons needed in the happens-before analysis. Goldilocks [51] combines locksets and happens-before in an unusual way by using locksets to efficiently track the happens-before relation. This precise, complicated analysis is embedded in a Java virtual machine to enable continuous monitoring of race conditions.

6.2.4 Predicting Data Races

Prior work focused on predictive data race detection is grounded in the context of happens-before race detection [32, 34, 35, 135, 137]. The main idea of this body of work is to consider which of the correct reorderings of critical sections would have triggered a race in a happens-before detector. The problem with a reorderings-based approach is that it requires exploring all reorderings of critical sections to determine which ones are correct and produce a race. This exploration is an expensive process: the space of possible reorderings is exponential and executions with races are often hard to discover. Such work is typically a hybrid of testing and model checking and does not achieve the polynomial complexity and scalability of the relation presented in Part IV. For instance, in the recent work by Said *et al.*[129], scalability relies on a modern SMT solver and an efficient encoding of the problem.

Among the body of work focused on dynamic race prediction, the jPredictor tool [36], based on the idea of sliced causality discussed in Section 6.1, is distinguished by explicitly producing a polynomial algorithm for race detection. Nevertheless, in order to do so, jPredictor abandons the general precision guarantees of the theory that underlies it. The main soundness theorem of the jPredictor work (which applies to more than race detection) states that every produced "consistent permutation" corresponds to a possible program execution. Nevertheless, "generating all the consistent permutations of a partial order is a #P-complete problem" [36]. To avoid an exponential search, jPredictor employs two shortcuts for the case of race detection. The first is avoiding a search of permutations: events are processed following the order of the original execution. For predictive power, jPredictor relies on an unsound definition of what constitutes a race (Definition 5 of [36]). This definition adapts (to sliced causality) the lockset criterion for race detection: two clashing, sliced causality-unordered accesses that occur without holding a common lock are considered to race. The second shortcut

is in the implementation, which performs a single slicing traversal of the trace, also resulting in unsoundness.

6.3 Atomicity

A variety of tools have been developed to detect atomicity violations, both statically and dynamically. Static analyses for verifying atomicity include type systems [64, 65, 165] and techniques that look for cycles in the happens-before graph [56] or def-use graphs [168]. Compared to dynamic techniques, static systems provide stronger completeness guarantees but typically involve trade-offs between precision and scalability.

Dynamic techniques analyze a specific executed trace at runtime. Artho *et al.* [11] developed a dynamic analysis tool to identify one class of "higher-level races" using the notion of view consistency. The Atomizer [61] uses Lipton's theory of reduction [96] to check whether steps of each transaction conform to a pattern guaranteed to be serializable. Wang and Stoller developed an alternative block-based algorithm [167] and also more precise commit-node algorithms [166]. Another method of detecting atomicity violations is by matching against 11 known violation patterns [157].

VELODROME [66] was the first dynamic atomicity analysis to find all errors on a particular trace without introducing false positives. This precise dynamic atomicity analysis uses a cycle-based algorithm with transactions as nodes and reports an error if and only if the observed trace is not serializable. VELODROME uses transactional happens-before graphs to detect cycles, which correspond directly to an atomicity violation. SIDETRACK (Part VI) uses a similar transactional happens-before algorithm to predict atomicity violations. Farzan and Madhusudan [57] provide space complexity bounds for a similar analysis.

Scheduling Atomicity Violations Sen and Park [115] have developed a tool called AtomFuzzer that attempts to schedule atomicity violations so that it is straightforward to determine if the atomicity violation represents a real error. When a thread is about to acquire the same lock a second time, AtomFuzzer pauses the thread (with probability 0.5), and tries to schedule another thread (which may cause an atomicity violation) first. If all threads are paused, one thread is chosen to continue. AtomFuzzer may miss some errors because it is not always possible (or desirable) to pause every thread as long as necessary at a vulnerable point, but may also shape a particularly useful execution trace through scheduling.

Software Transactional Memory Software transactional memory [92] is a programming model proposed as an alternative to lock-based concurrency, where the run-time system ensures the atomic execution of transactions. This approach involves forcing programmers to use a new programming paradigm, and does not easily apply to legacy code. Some recent work has been devoted to addressing the semantic difficulties of software transactional memory [140] and making it compatible with lock-based programming [141, 156]. Software transactional memory enforces the atomic execution of transactions, while atomicity uses program analysis to guarantee that atomic blocks always execute as if in such a transaction.

6.3.1 Predicting Atomicity Violations

Several recent papers have addressed predicting atomicity violations (in particular) from an observed trace. Most predictive approaches are based on leveraging static control- and data-flow information and suffer from efficiency concerns. Farzan and Madhusudan [58] predict runs from program models based on *profiles* and check serializability of the predicted runs.

Another interesting predictive approach consists of permissive reordering mod-

els on executions.[2] Past work [59, 86] assumes that threads only communicate via holding locks, and not by writing to shared memory locations. Any trace that holds the same locks in the same order of nesting is considered an appropriate generalization of the observed behavior in that predictive model. The Penelope [142] system then adds soundness back by trying to reproduce the predicted atomicity violation through changes to the scheduling of a real execution. The published experiment numbers imply that their approach does not scale at or near the level of the algorithms presented in Part IV, V, and VI and can suffer from high costs in small yet complex executions. Combinations of approaches along these lines should be interesting future work.

6.4 Cooperability

The notion of cooperability, where code executes *as if* context switches happen only at specific yield points, was partly inspired by recent work on automatic mutual exclusion, which proposes ensuring mutual exclusion by default [1, 3, 77]. The relationship between cooperability and automatic mutual exclusion is analogous to that between atomicity and transactional memory. Cooperative semantics have been explored in various settings; for example in high scalability thread packages [159] and as alternative models for structuring concurrency [7, 10, 23]. There is also a type and effect system for cooperability [169, 170], a dynamic analysis tool [172], and a user evaluation [128].

6.5 Determinism

Checking for semantic determinism is pretty tricky. For the most part, multi-threaded determinism checkers are focused on verifying that alternate schedules do not change behaviour, instead of dealing with things like random number generation.

[2] These models can be more permissive than the correctly-reorders relation (Definition 7 in Chapter 11) or the approach taken by SideTrack (Part VI).

Dynamic checkers for determinism verify that the observed trace could not be rearranged to change behaviour but are limited by the observed trace. SingleTrack [126], a deterministic parallelism analysis tool, checks serializability of multi-threaded transactions that are also verified to be free of internal conflicts (thus guaranteeing determinism at the transaction level). This work can be viewed as an extension of atomicity, and in fact reduces to atomicity detection if all transactions are single-threaded. If there is only one transaction, then SingleTrack acts as a determinism checker.

Another issue with checking for determinism is defining when exactly two executions have the "same" behaviour. For example, rounding errors in arithmetic operations involving floating point numbers may cause multiple executions of the same code to end up with very slightly different results. Burnim *et al.* present an assertion-based mechanism for checking determinism, where *bridge assertions* can be used to relate different executions of the same program [27, 28].

Enforcement schemes for determinism, which ensure consistent behaviour at the expense of some efficiency, also exist [114], as well as type and effect systems [18]. There is the potential for practical operating-systems-based enforcement of determinism [13] or even for hardware-based enforcement of determinism [45].

6.6 Deadlock Detection

The main way to avoid deadlocks is to have a consistent order for acquiring locks. Type systems exist for specifying partial orders over locks in a program [24]. There are also a few well-known static analysis systems for deadlock detection. For example, RacerX [53] constructs a graph of lock orderings and checks for cycles. An award-winning recent paper on static deadlock detection [108] abstracts six necessary conditions for deadlock freedom, and then uses separate analyses for each of those six

conditions.

Dynamic analyses for deadlock detection reduce (but do not eliminate) false positives with a dynamic lock ordering graph [16]. Potential deadlocks reported by dynamic analysis can also be used to drive the scheduler towards deadlock manifestation [83].

Part III

Technical Background

Chapter 7

Semantics

We begin by formalizing the notion of multithreaded program traces, which serve as the input to our dynamic analyses. A program consists of a number of concurrently executing threads that manipulate shared variables $x \in \mathit{Var}$ and locks $m \in \mathit{Lock}$. Each thread has a thread identifier $t \in \mathit{Tid}$. A *trace* α captures an execution of a multithreaded program by listing the sequence of operations performed by the various threads. We use the shorthand notation $a \in \alpha$ to convey that operation a appears in the trace α. The set of operations that thread t can perform include:

- $r(t, x, v)$ and $w(t, x, v)$, which read and write a value v from variable x; and
- $acq(t, m,)$ and $rel(t, m)$, which acquire and release a lock m.

Our implementations also support a variety of additional operations such as fork, join, wait, notify, and access to volatile variables. We omit the value from read and write operations if it is not relevant.[1]

This lower level abstraction of traces allows us to ignore objects and their composition; more complicated concurrency bugs boil down to short sequences of these basic operations. We refer to the actual trace perceived by our analysis tool as the

[1]When illustrating program traces, we adopt the convention from Figure 1.7 where the thread identifier is above instead of inside the operations by that thread.

observed trace. In addition to the observed trace, many other *feasible* traces exist for a given program. It is not generally possible to predict all feasible traces for a program without additional static information; for example, the observed trace may not have complete code coverage for the source program. The notion of trace feasibility directly relates to the precision of predictive dynamic analysis tools. For example, if a precise predictive race detector flags a race on a trace, then a feasible trace exists that contains a happens-before race. Note that it may be possible to prove that a feasible trace exists satisfying a certain property without actually producing such a feasible trace. If we are able to predict, given an observed trace, that a potential concurrency error (such as a data race or atomicity violation) occurs on another feasible trace, we say there is a *predictable* concurrency error.

7.1 Trace Evaluation

For illustration purposes, this section contains a further formalization of trace evaluation. Trace evaluation, a concept from operational semantics, clarifies how to figure out the program state after a trace is executed. Each operation in a trace has a unique index into that trace. For example, to distinguish between two identical reads of `x` in a trace, this index would be useful. A trace is a sequence of operations such that the indices start at 0 and the index of each operation is 1 greater than the index of the previous operation. To reduce notational clutter, these indices are elided.

The meaning of each operation a is defined by the *trace evaluation* relation $\Sigma \rightarrow^a \Sigma'$, where the program state Σ maps variables to values, and also maps each lock to a thread identifier, or to $\perp$ if that lock is not held by any thread. The relation $\Sigma \rightarrow^a \Sigma'$ describes how the state Σ is updated to Σ' by each operation a, as formalized by the rules in Figure 7.1. This figure uses standard operational semantics terminology where events above the line are preconditions to results below the line. For example, the

rule [ACQ] states that no thread holds the lock directly before the acquire operation and the acquiring thread holds the lock directly afterwards. The notation $Tid_\perp$ refers to the union of the space of thread identifiers and $\perp$, which is used when no thread currently holds a particular lock. The extension of this relation from individual operations a to traces (or operation sequences) α is straightforward.

Trace evaluation starts in the initial state Σ_0 where all variables are zero-initialized and all locks are initially not held by any thread:

$$\Sigma_0 = (\lambda x.0) \cup (\lambda m.\perp)$$

We also assume a helper function for extracting the thread identifier from an operation:

$$tidof : Operation \to Tid$$

And a helper function for extracting the (implicit) index from a trace operation:

$$idof : Operation \to \mathbb{N}$$

7.2 Well-Formed Traces

We are concerned with the traces produced from an actual run of a source program. Our assumptions include the well-nesting of locks, as well as standard lock semantics.

In a well-formed trace, every release operation has a *matching* acquire operation by the same thread earlier in the trace, with no intervening acquires of that lock. This restriction is also captured in Figure 7.1. To explicate this relationship between matching acquire and release operations, we sometimes annotate each acquire/release

Figure 7.1: Multithreaded Program Traces

Domains

$$
\begin{array}{rcll}
t, u & \in & \mathit{Tid} & \\
x, y & \in & \mathit{Var} & \\
v & \in & \mathit{Value} & \\
l, m & \in & \mathit{Lock} & \\
a, b & \in & \mathit{Operation} & ::= \quad r(t,x,v) \\
 & & & \mid w(t,x,v) \\
 & & & \mid \mathit{acq}(t,m) \\
 & & & \mid \mathit{rel}(t,m) \\
\alpha, \beta, \gamma & \in & \mathit{Trace} & = \quad \mathit{Operation}^* \\
\Sigma & \in & \mathit{State} & = \quad (\mathit{Var} \mapsto \mathit{Value}) \cup (\mathit{Lock} \mapsto \mathit{Tid}_\perp)
\end{array}
$$

Trace Evaluation

$$
[\textsc{read}] \quad \frac{\Sigma(x) = v}{\Sigma \rightarrow^{r(t,x,v)} \Sigma} \qquad [\textsc{write}] \quad \frac{}{\Sigma \rightarrow^{w(t,x,v)} \Sigma[x := v]}
$$

$$
[\textsc{acq}] \quad \frac{\Sigma(m) = \perp}{\Sigma \rightarrow^{acq(t,m)} \Sigma[m := t]} \qquad [\textsc{rel}] \quad \frac{\Sigma(m) = t}{\Sigma \rightarrow^{rel(t,m)} \Sigma[m := \perp]}
$$

pair with a unique label k; so that $acq^k(t, m)$ is matched with $rel^k(t, m)$, and vice versa.

We also assume locks are not re-entrant.[2] A trace is *closed* if every acquire operation has a matching release. A *synchronized block* is a sequence of operations by a thread starting with an acquire operation and ending with a matching release. We assume that traces are *well-nested*, in that any synchronized block has no locking side effect.

Definition 1 (Well-Formed Trace)**.** A *well-formed trace* is a total order of events such that

1. Acquisition of a lock is not followed by another acquisition of the same lock without

[2]Our implementations support re-entrant locking operations but filter them out before further analysis.

an intervening matching lock release.

2. Synchronization blocks are well-nested. More explicitly, if an acquisition of lock l_2 is performed after an acquisition of lock l_1 by the same thread and before l_1's matching release then the matching release of l_1 cannot appear before the matching release of l_2 does.

In future sections, we refer to well-formed traces simply as traces.

7.3 Conflicts and Happens-Before

Before defining the happens-before relation, we review the definition of clash and conflict along with some related relations over the operations in a trace.
We begin by assuming a helper relation:

- the binary relation $\asymp$ (read *clashes*). Two events by different threads clash if they both access the same variable and one of the operations is a write.

We also introduce four basic orders over the operations in a trace: total order (TO), which is a total order over all operations; program order (PO), which orders operations by the same thread; release-acquire order (RA), which relates each release operation on a lock with every subsequent acquire on that lock; and communication order (CO), which orders clashing operations.

$TO^{\alpha} \stackrel{\text{def}}{=} \{(a, b) \mid a \in \alpha,\, b \in \alpha,\, idof(a) <= idof(b)\}$[3]

$PO^{\alpha} \stackrel{\text{def}}{=} \{(a, b) \in TO^{\alpha} \mid \text{tid of } a = \text{tid of } b\}$

$RA^{\alpha} \stackrel{\text{def}}{=} \{(r, a) \in TO^{\alpha} \mid r = rel(t, m) \text{ and } a = acq(u, m)\}$

$CO^{\alpha} \stackrel{\text{def}}{=} \{(a, b) \in TO^{\alpha} \mid a \asymp b\}$

[3]These orders are defined using set builder notation from ISO 31-11.

Figure 7.2: Simple trace example γ (on left; from Figure 1.9) and η (on right; from Figure 1.8).

Thread 1	*Thread 2*
a: `w(x)`	
b: `acq(l)`	
c: `rel(l)`	
d:	`acq(l)`
e:	`rel(l)`
f:	`w(x)`

Thread 1	*Thread 2*
u:	`acq(l)`
v:	`rel(l)`
w:	`w(x)`
x: `w(x)`	
y: `acq(l)`	
z: `rel(l)`	

For example, consider the traces γ and η in Figure 7.2. Figure 7.3 lists all the TO, PO, RA and CO edges over the two example traces. We write $a <^{\alpha}_{\mathcal{X}} b$ to abbreviate $(a, b) \in \mathcal{X}^{\alpha}$ for some ordering relation $\mathcal{X}$ over trace α, and we omit the superscript α when it is clear from the context.

The happens-before relation for a trace α is a partial order over the operations in α that characterizes which operations in a trace may enable or influence subsequent operations in the trace. This characterization leverages a notion of conflicting operations that cannot be reordered without potentially impacting the behaviour of a trace. More formally, two operations in a trace *conflict* if they satisfy one of the following conditions:[4]

- **Lock conflict:** the operations acquire or release the same lock.
- **Program order conflict**: the operations are performed by the same thread.

The happens-before relation is based on the orderings of conflicting operations. Adjacent non-conflicting operations are said to commute, as swapping their order in a trace will not influence the behaviour of the trace.

Definition 2 (Happens-before)**.** The *happens-before relation* for a trace α is the transitive closure of the union of the program order and release-acquire order edges.

$$<^{\alpha}_{HB} \stackrel{\text{def}}{=} (PO^{\alpha} \cup RA^{\alpha})^{*}$$

[4]Sometimes, the phrase *communication conflict* (or communication-order conflict) is used to refer to a clash; note that CO edges do not form part of the happens-before relation.

Figure 7.3: Various relations over traces from Figure 7.2.

TO^γ	(a,a), (a,b), (a,c), (a,d), (a,e), (a,f), (b,b), (b,c), (b,d), (b,e), (b,f), (c,c), (c,d), (c,e), (c,f), (d,d), (d,e), (d,f), (e,e), (e,f), (f,f)	TO^η	(u,u), (u,v), (u,w), (u,x), (u,y), (u,z), (v,v), (v,w), (v,x), (v,y), (v,z), (w,w), (w,x), (w,y), (w,z), (x,x), (x,y), (x,z), (y,y), (y,z), (z,z)
PO^γ	(a,a), (a,b), (a,c), (b,b), (b,c), (c,c), (d,d), (d,e), (d,f), (e,e), (e,f), (f,f)	PO^η	(u,u), (u,v), (u,w), (v,v), (v,w), (w,w), (x,x), (x,y), (x,z), (y,y), (y,z), (z,z)
RA^γ	(c,d)	RA^η	(v,y)
CO^γ	(a,f)	CO^η	(w,x)
$<^\gamma_{HB}$	TO^γ	$<^\eta_{HB}$	(v,y), (u,u), (u,v), (u,w), (v,v), (v,w), (w,w), (x,x), (x,y), (x,z), (y,y), (y,z), (z,z), (u,y), (u,z), (v,z)

Figure 7.3 lists all edges of the happens-before relation for the traces from Figure 7.2. Notice that `x` is involved in a happens-before race in η, but not in γ. An alternative formulation of happens-before is below:

Definition 3 (Happens-before)**.**

- Events by the same thread are ordered as they appear.

  $a <^\alpha_{HB} b$ if $(a,b) \in PO^\alpha$

- Releases and acquisitions of the same lock are ordered as they appear.

  $a <^\alpha_{HB} b$ if $(a,b) \in RA^\alpha$

- $<_{HB}$ is closed under composition with itself.

  $<_{HB} = (<_{HB} \circ <_{HB})$

Two traces are *equivalent* if one can be obtained from the other by repeatedly swapping adjacent non-conflicting (commuting) operations. Equivalent traces yield the same happens-before relation, and exhibit equivalent behavior.

Definition 4 (HB-race)**.** A trace α has a *happens-before race condition* if there exists two clashing operations $a, b \in \alpha$ that are not ordered by happens-before.

This definition of a happens-before race condition captures the notion that there is no synchronization between two clashing accesses on a particular variable. Hence those clashing accesses could have been scheduled in either order, introducing (probably unintentional or erroneous) non-determinism into the program's behavior.

Chapter 8

Vector Clocks

SideTrack and Embracer use vector clocks [103] to identify concurrent acquire operations within a trace by compactly representing the happens-before relation. A vector clock maps thread identifiers to integer clocks:

$$K \stackrel{\text{def}}{=} \mathit{Tid} \to \mathbb{N}$$

If k is the vector clock for an acquire operation in a trace, then $k(t)$ identifies the operations of thread t that happen-before that acquire.

Vector clocks are partially-ordered ($\sqsubseteq$) in a point-wise manner, with an associated join operation ($\sqcup$) and minimal element ($\bot$). In addition, the helper function inc_t increments the t-component of a vector clock. For all vector clocks k, k_1, and k_2:

$$\begin{array}{rcl} k_1 \sqsubseteq k_2 & \text{iff} & \forall t.\ k_1(t) \leq k_2(t) \\ k_1 \sqcup k_2 & = & \lambda t.\ \mathit{max}(k_1(t), k_2(t)) \\ \bot & = & \lambda t.\ 0 \\ \mathit{inc}_t(k) & = & \lambda u.\ \texttt{if}\ u = t\ \texttt{then}\ k(u) + 1\ \texttt{else}\ k(u) \end{array}$$

8.1 Simple Happens-Before Race Detection

As an illustration of vector clocks, we formalize a happens-before detector as an online analysis based on an analysis state $\phi = (\mathbb{C}, \mathbb{R}, \mathbb{W}, \mathbb{U})$, where:

- $\mathbb{C} : \mathit{Tid} \to K$ records the vector clock of the current operation by each thread;
- $\mathbb{R} : \mathit{Var} \to K$ records the vector clock of the last read of each shared variable;
- $\mathbb{W} : \mathit{Var} \to K$ records the vector clock of the last write of each shared variable and volatile variable; and
- $\mathbb{U} : \mathit{Lock} \to K$ records the vector clock of the last unlock of each lock.

In the initial state ϕ_0, all vector clocks are initialized to $\bot$, except each $\mathbb{C}_t$ starts at $\mathit{inc}_t(\bot)$ to reflect that the first steps by different threads are not ordered.

$$\phi_0 \quad = \quad (\lambda t.\ \mathit{inc}_t(\bot),\ \lambda x.\ \bot,\ \lambda x.\ \bot,\ \lambda m.\ \bot)$$

Figure 8.1 formalizes how the HB state is updated for each operation a of the target program's trace, via the relation $\phi \Rrightarrow^a \phi'$. Each of the analysis rules in Figure 8.1 consists of one or more antecedents (above the line) and a single consequent (below the line). When implementing these rules, it is useful to break down the antecedents into *checks*, and *updates*. Checks indicate when the analysis should flag an error. Updates describe how the next analysis state is obtained.

- The rule [HB READ] signals a happens-before race when the prior operation of the reading thread is unrelated by happens-before with the previous write operation for that variable; a read operation may commute with other read operations unrelated by happens-before and still exhibit the same behavior. Here, $\mathbb{C}$ is a map, $\mathbb{C}_t$ is an abbreviation for $\mathbb{C}(t)$, and $\mathbb{C}[t := k]$ denotes a map that is identical to $\mathbb{C}$ except that it matches t to k.

Figure 8.1: Happens-Before Race Detection Algorithm

$$\frac{\mathbb{W}_x \not\sqsubseteq \mathbb{C}_t \implies \texttt{race} \quad \mathbb{R}' = \mathbb{R}[x := \mathbb{R}_x[t := \mathbb{C}_t(t)]]}{(\mathbb{C}, \mathbb{R}, \mathbb{W}, \mathbb{U}) \Rrightarrow^{r(t,x,v)} (\mathbb{C}, \mathbb{R}', \mathbb{W}, \mathbb{U})} \quad [\text{HB READ}]$$

$$\frac{\mathbb{R}_x \not\sqsubseteq \mathbb{C}_t \implies \texttt{race} \quad \mathbb{W}_x \not\sqsubseteq \mathbb{C}_t \implies \texttt{race} \quad \mathbb{W}' = \mathbb{W}[x := \mathbb{W}_x[t := \mathbb{C}_t(t)]]}{(\mathbb{C}, \mathbb{R}, \mathbb{W}, \mathbb{U}) \Rrightarrow^{w(t,x,v)} (\mathbb{C}, \mathbb{R}, \mathbb{W}', \mathbb{U})} \quad [\text{HB WRITE}]$$

$$\frac{\mathbb{C}' = \mathbb{C}[t := \mathbb{C}_t \sqcup \mathbb{U}_m]}{(\mathbb{C}, \mathbb{R}, \mathbb{W}, \mathbb{U}) \Rrightarrow^{acq(t,m)} (\mathbb{C}', \mathbb{R}, \mathbb{W}, \mathbb{U})} \quad [\text{HB ACQUIRE}]$$

$$\frac{\mathbb{U}' = \mathbb{U}[m := \mathbb{C}_t] \quad \mathbb{C}' = \mathbb{C}[t := inc_t(\mathbb{C}_t)]}{(\mathbb{C}, \mathbb{R}, \mathbb{W}, \mathbb{U}) \Rrightarrow^{rel(t,m)} (\mathbb{C}', \mathbb{R}, \mathbb{W}, \mathbb{U}')} \quad [\text{HB RELEASE}]$$

- Similarly, the rule [HB WRITE] signals a happens-before race when the prior operation of the writing thread is unrelated by happens-before with the previous read or write operation for that variable.

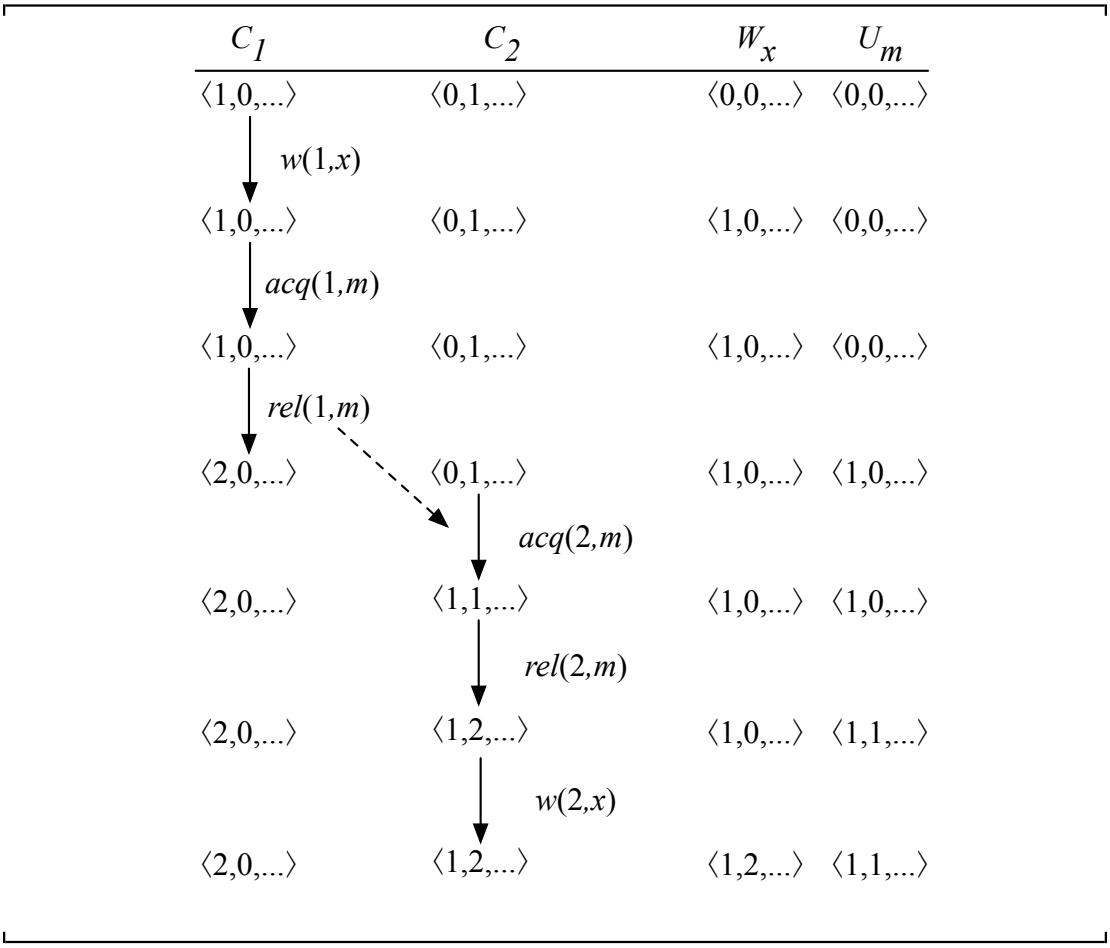
- An acquire operation must happen after any prior release operations; the rule [HB ACQUIRE] joins the current vector clock with the time of the previous release.

- The rule [HB RELEASE] records when the lock is released and increments the current vector clock.

Figure 8.2 shows the relevant portions of the analysis state, and how these state components are updated, as each operation in the trace γ on the left of Figure 7.2

is processed. The first operation updates W_x with the write time of `x`.[1] The first acquire does not alter any of the state components since U_m is still $\perp$. The subsequent release by Thread 1 updates U_m with the time of the release and then increments the clock for Thread 1. The acquire by Thread 2 updates C_2 to reflect the time of the last release (by Thread 1). The subsequent release by Thread 2 updates U_m with the time of the release and then increments the clock for Thread 2. When Thread 2 writes `x`, $W_x \sqsubseteq C_2$ and so no race is reported.

Figure 8.2: Happens-Before Vector Clock Example



[1] R_x is left off since the trace does not contain any reads.

Chapter 9

RoadRunner

RoadRunner is a framework, written in Java and developed by Cormac Flanagan and Stephen Freund, designed for developing dynamic analyses for multi-threaded programs [63]. RoadRunner supports chaining of tools, so that multiple analyses can be run concurrently. RoadRunner also allows direct comparison between tools in an apples-to-apples way.

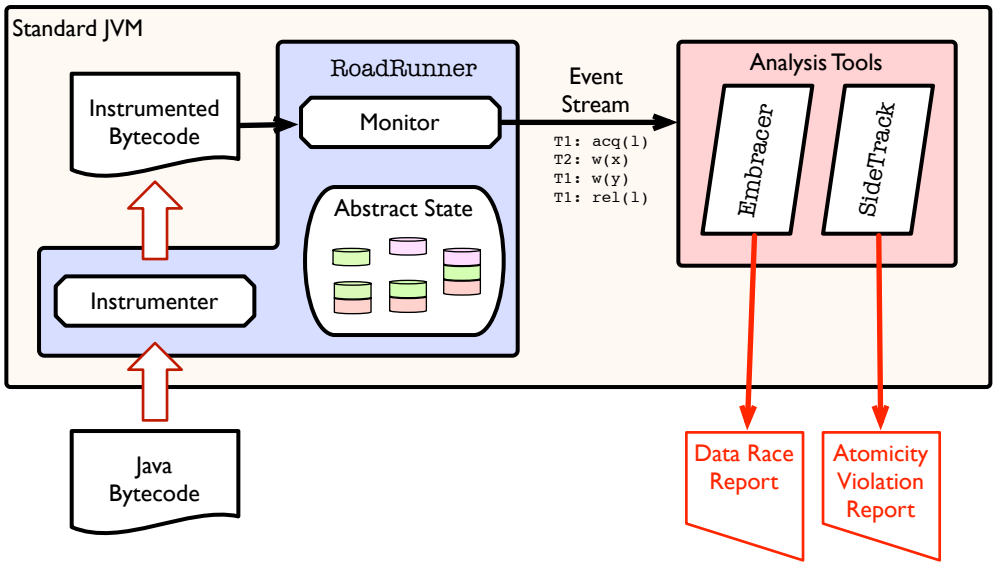
RoadRunner instruments the target bytecode of a program during load time and provides a clean API for analysis tools to run over an event stream. The instrumentation code generates a stream of events for lock acquires and releases, field and array accesses, method entries and exits, etc. Re-entrant lock acquires and releases are redundant and are filtered out. Tools implemented in RoadRunner, such as Embracer and SideTrack, process this event stream as it is generated. See Figure 9.1 for an overview of this structure. In this figure, Java bytecode is instrumented by RoadRunner as it is loaded. This instrumented bytecode produces an event stream which is fed to analysis tools.

RoadRunner enables analysis tools to attach instrumentation state to each thread, lock object, and data memory location used by the target program. This feature is represented by the "Abstract State" in Figure 9.1. Tool-specific event handlers update

the instrumentation state for each operation in the observed trace and report errors when appropriate. These handlers can be implemented by simply defining methods for particular events of interest. These methods may leverage custom instrumentation state.

We have implemented a number of tools for the work in this dissertation, including many unpublished prototypes. RoadRunner proved excellent for rapidly prototyping new dynamic analysis algorithms. We found the use of RoadRunner to be good for research methodology: direct implementation comparisons between two algorithms leads to meaningful and substantial experimental results. Furthermore, RoadRunner is open source and widely accessible.

Figure 9.1: Diagram showing the overall structure of how RoadRunner works with tools.



Part IV

CP Relation

Chapter 10

Introduction

Part IV is focused on the causally-precedes (CP) relation: a novel relation over the operations in a trace that is less restrictive than the happens-before relation. A precise race detector can be built from this relation by reporting clashing operations in a trace that are unordered by the CP relation as racing. Such a detector is of polynomial complexity and does not introduce false positives. This work was originally presented at the 2012 Symposium on Principles of Programming Languages (POPL) [139].

The problem with plain happens-before race detection (Section 6.2.1 and Section 7.3) is that it can miss many races due to accidental HB edges. The original definition of happens-before by Lamport [91] was in the context of distributed systems, with an HB edge introduced for explicit inter-process communication. However, lock synchronization does not induce the same hard ordering as explicit communication. A lock-based critical section can often be reordered with others, as long as lock semantics (*i.e.* mutual exclusion) is preserved.

Consider the code example shown in Figure 10.1. In this example, the class `PolarCoord` has two fields, `radius` and `angle`, protected by the object lock `this`. The `count` field tallies the number of accesses to `radius` and `angle`, and the `main` method forks two concurrent threads. This program has a race condition on `count`; unfortunately,

Figure 10.1: Example Program `PolarCoord`

```
1  class PolarCoord {
2    int radius, angle;
3    int count; // counts accesses
4
5    static PolarCoord pc = new PolarCoord();
6
7    void setRadius(int r) {
8      count++;
9      synchronized(this) { radius = r; }
10   }
11
12   int getAngle() {
13     int t;
14     synchronized(this) { t = angle; }
15     count++;
16     return t;
17   }
18
19   public static void main(String[] args){
20     fork { pc.setRadius(10); }
21     fork { pc.getAngle(); }
22   }
23 }
```

precise race detectors such as FastTrack [62] or djit$^+$ [119] fail to detect this race condition on 94% of test runs.

Figure 10.2(A) illustrates the essence of the problem by showing the trace that the HotSpot JVM typically generates for the program of Figure 10.1, with no overlap between the executions of the two threads. For this trace, a happens-before race detector would not find a race on `count`, since the lock release by Thread 1 *happens-before* the lock acquire of Thread 2, thereby masking the lack of synchronization between the accesses to `count`. In contrast, Trace B presents a different scheduling where there is clearly a race on `count`. By inspection, we are able to predict from Trace A that a race condition *could* occur as in Trace B; we say that Trace A has a *predictable* race.

In the following chapters, we define a novel relation, *causally-precedes* (CP), by analogy to happens-before, to detect such races. A race occurs if two clashing

Figure 10.2: Example Traces for the `PolarCoord` Program.

Thread 1	*Thread 2*
`r(count)`	
`w(count)`	
`acq(this)`	
`  w(radius)`	
`rel(this)`	
	`acq(this)`
	`  r(angle)`
	`rel(this)`
	`r(count)`
	`w(count)`

Tool	Report
Happens-Before:	"no race"
Causally-Precedes:	"race"

(A) Predictable race condition

Thread 1	*Thread 2*
	`acq(this)`
	`  r(angle)`
	`rel(this)`
`r(count)`	
`w(count)`	
	`r(count)`
	`w(count)`
`acq(this)`	
`  w(radius)`	
`rel(this)`	

Tool	Report
Happens-Before:	"race"
Causally-Precedes:	"race"

(B) Happens-before race

operations are not CP-ordered. Unlike prior precise race detectors, a CP-based race detector can detect predictable race conditions as in Figure 10.2(A). The essence of detecting this predictable race is that the critical section of Thread 2 has received no information that can reveal whether the critical section of Thread 1 has already executed or not. More precisely, reordering events as in trace B (thus exposing an HB race) maintains the property that *all read operations return exactly the same values as in the original execution*—we call this a *correct reordering* of the observed behavior. A correctly reordered execution is just as feasible as the observed one.

CP-based race detection offers the first sound yet scalable technique for predictive race detection.[1] Specifically, CP weakens the HB order while still maintaining soundness. CP race detection results are not complete: examining all correct reorderings of the original trace would necessitate an exponential search. Consequently, a CP-based detector may miss some races. Nevertheless, it is guaranteed to detect a (non-strict) superset of the observed happens-before races and to only give warnings for true races.

In the following chapters, we formally define the CP relation and demonstrate with numerous examples why it is not easy to weaken HB while remaining precise. For reference, a replication of the soundness proof for CP (originally presented in POPL 2012 [139]) can be found in Appendix B and preliminary experimental results for a CP-based race detector can be found in Appendix A.

[1]Prior work focused on predictive data race detection extends happens-before race detection with a hybrid of testing and model checking [32, 34, 35, 135, 137]. See Section 6.1 and 6.2.4 for a more detailed discussion on the related work and its limitations.

Chapter 11

CP Relation

We now formally define the causally-precedes relation.

Definition 5 (Causally Precedes)**.** Causally precedes ($<_{CP}$) is the smallest relation such that:

a) $<_{CP}$ has a release-acquire edge between critical sections over the same lock that contain clashing events.

In other words, $rel^k(t,l) <_{CP} acq^h(u,l)$ if there are operations a and b such that:

- $a \asymp b$
- $acq^k(t,l) <_{\text{PO}} a <_{\text{PO}} rel^k(t,l)$
- $acq^h(u,l) <_{\text{PO}} b <_{\text{PO}} rel^h(u,l)$
- $rel^k(t,l) <_{\text{TO}} acq^h(u,l)$

b) $<_{CP}$ has a release-acquire edge between critical sections over the same lock that contain CP-ordered events. These events can be lock acquisition or release operations, and not necessarily internal events in the critical section.

Because of Rule (c), below, this condition turns out to be equivalent to the seemingly weaker "releases and acquisitions of the same lock are ordered if the beginning of one critical section is CP-ordered with the end of the other"—since there

is an HB order between the start of a critical section and every internal event, as well as all internal events and the end of a critical section. In other words, $rel^k(t,l) <_{CP} acq^h(u,l)$ if $k \neq h$ and $acq^k(t,l) <_{CP} rel^h(u,l)$

c) CP is closed under left and right composition with HB.

$<_{CP} = (<_{HB} \circ <_{CP}) = (<_{CP} \circ <_{HB})$

Thread creation and joining can be added straightforwardly, as explicit causally-precedes edges; for simplicity, we do not discuss these events in the examples. Note that $<_{CP}$ is a subset of the happens-before relation. Inspecting the three cases of the $<_{CP}$ definition, we see that all $<_{CP}$ edges produced by the first two rules are release-acquire edges on the same lock and so are also $<_{HB}$ edges. The third rule then states that CP is closed under composition with HB, which still produces a subset of the HB edges, since HB is transitively closed. It is similarly easy to see that CP is transitive.

Every CP race (and hence every HB race) is also a lockset race, since the absence of a CP edge between clashing accesses to a location means that there is no consistently held protecting lock for that location. Consequently, a lockset-based race detector would detect all races detected by CP,[1] and may also report many additional warnings. In practice, many of these extra warnings are false alarms that do not correspond to actual races.

Definition 6 (CP-Race)**.** We define a *race* (or *CP-race* when we need to distinguish from happens-before races) to be a pair of clashing events that are not CP-ordered in either direction.

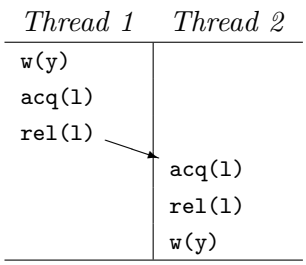
[1]Note that the Eraser algorithm [131], which incorporates lockset-based reasoning, is also slightly incomplete in how it reasons about thread-local data, and so may miss some real HB or CP races.

11.1 Illustration

There are a few aspects of the definition of CP that should be emphasized for clarity. Probably most important among them is that CP is not a reflexive relation. (If it were, Rule (c) would make CP equal to HB.) Consequently, CP is also not a partial order.

Recall that we want CP to retain only some of the HB edges in a way that captures which clashing events could have happened simultaneously in a reordered but certain-to-be-feasible execution. Consider again the example of Figure 10.2 and observe that the shown HB edge is not a CP edge. None of the events shown are CP-ordered—*i.e.* they constitute a CP-race (and a predictable race). The same occurs in the execution of Figure 11.1.

Figure 11.1: Another example of a certain race not reported by HB.

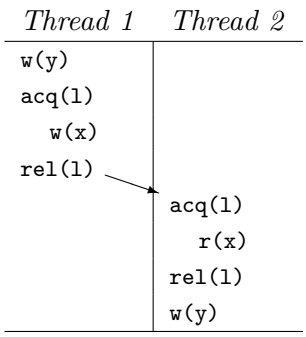


In both cases, the critical sections do not contain clashing events, which would order them per Rule (a) of the CP definition. In contrast, Figure 11.2 incurs no CP race report: the HB release-acquire edge is also a CP edge (per Rule (a)) and Rule (c) can then be used to CP-order the two operations on variable `y`.

Indeed, Rule (a) of the CP definition is almost inevitable. The ordering of critical sections containing clashing events is a key part of how information flows through a trace; changing this ordering could cause a trace to diverge.

Definition 7 (Correctly Reorders)**.** We say that an execution ex' correctly reorders

Figure 11.2: Example with no predictable race.



(CR) another execution ex (also written $ex' =_{CR} ex$) iff ex' is a total order over a subset of the events of ex that:

- contains a prefix of the events of every thread in ex and respects program order, *i.e.* if an event e in ex appears in ex' then all events by the same thread that precede e in ex also appear in ex' and precede e.
- for every read event that appears in ex', the most recent write event of the same variable in ex' is the same as the most recent write event of the same variable in ex.

The definition of CR matches an intuition for feasible alternative executions: if every value read is the same as in the observed execution, then the alternative is certainly also feasible.[2]

In this light, Rule (a) from the definition of CP is intuitively clear: as far as later events are concerned, two clashing events have to occur in the same order in every correctly-reordered execution. Thus, clashing events induce a hard ordering dependency, if it is certain that they do not constitute a race (in this case, because they are protected by a common lock). Since two critical sections over the same lock have

[2]This pairing of a read with the most recent write event implicitly introduces sequential consistency as an assumption. Nevertheless, this is not a constraint: Every HB race is a CP race. In case no HB races are observed for an execution, a relaxed memory model yields sequentially consistent behavior, thus our assumption is valid.

to be ordered in their entirety (*i.e.* all events of one have to precede all events of the other) the ordering constraint on clashing events becomes an ordering constraint on the entire critical section containing them. The same reasoning applies to Rule (b) of the CP definition: if two events internal to respective critical sections are CP-ordered, then the entire critical sections are also CP-ordered.

Rule (c) is the most interesting aspect of the CP definition. The rule is both very conservative and surprisingly weak, in different ways. Intuitively, Rule (c) is directly responsible for the soundness of CP: once some evidence of inevitable event ordering is found, all earlier and later events in an HB order automatically maintain their relative order. This conservative aspect of the Rule is necessary for ensuring that a CP race truly indicates that the events could have been concurrent. At the same time, Rule (c) is quite weak. Consider three events e_1, e_2, and e_3. It could be that $e_1 <_{CP} e_2$, $e_2 <_{HB} e_3$, and consequently $e_1 <_{CP} e_3$. This still does *not* mean that $e_2 <_{CP} e_3$, even though the HB order between e_2 and e_3 is what allows e_1 to CP e_3. This aspect is what allows CP to identify the possibility of predictable races (as in e_2-does-not-causually-precede-e_3) even when it assumes conservatively that certain reorderings are not possible, in order to maintain soundness.

Chapter 12

Example Traces

In this chapter, we discuss the subtleties of CP through examples of hard-to-reason-about executions. For each example, it is instructive for the reader to consider independently whether there is a predictable race or not. Reasoning about concurrent executions is quite hard: even concise examples require exhaustive examination of a large number of possible schedulings or complex formal reasoning to establish ordering properties. In the following examples, we expressed the constraints as symbolic inequalities with disjunctions (*e.g.* "this event is either before that or after the other") and proved manually they were unsatisfiable.

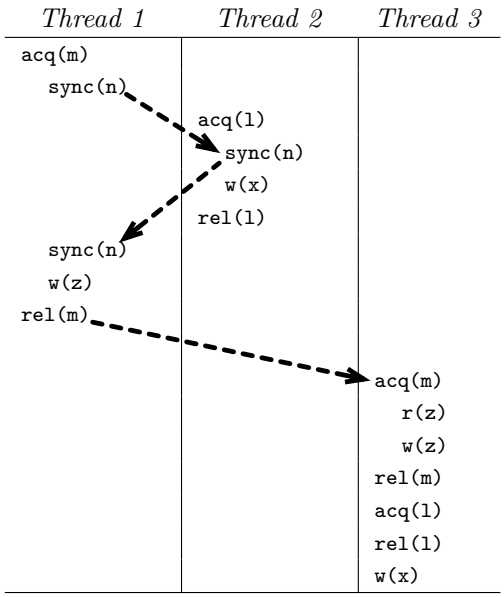
These example traces were developed over a multi-year period to expose errors in many prior failed attempts to create a precise relation that is less restrictive than HB. They are the key examples that led to the current CP definition, and represent an acid test for future such relations.

12.1 Example Highlighting Second Rule in CP Definition

Figure 12.1 contains a first example that suggests why sound predictive race detection is hard in the presence of many threads and nested locks. We use the shorthand

`sync(`*lock*`)` for a sequence of events that induces an inevitable ordering with other identical `sync` sequences. For example, `sync(n)` can be short for `acq(n); r(nVar); w(nVar); rel(n)`. For ease of reference, CP edges produced by Rule (a) of the CP definition are shown in the figure, as dotted arrows. The example from Figure 12.1 necessitated the addition of rule (b) to the CP definition.

Figure 12.1: No race between the two writes to `x` in any correctly reordered execution.



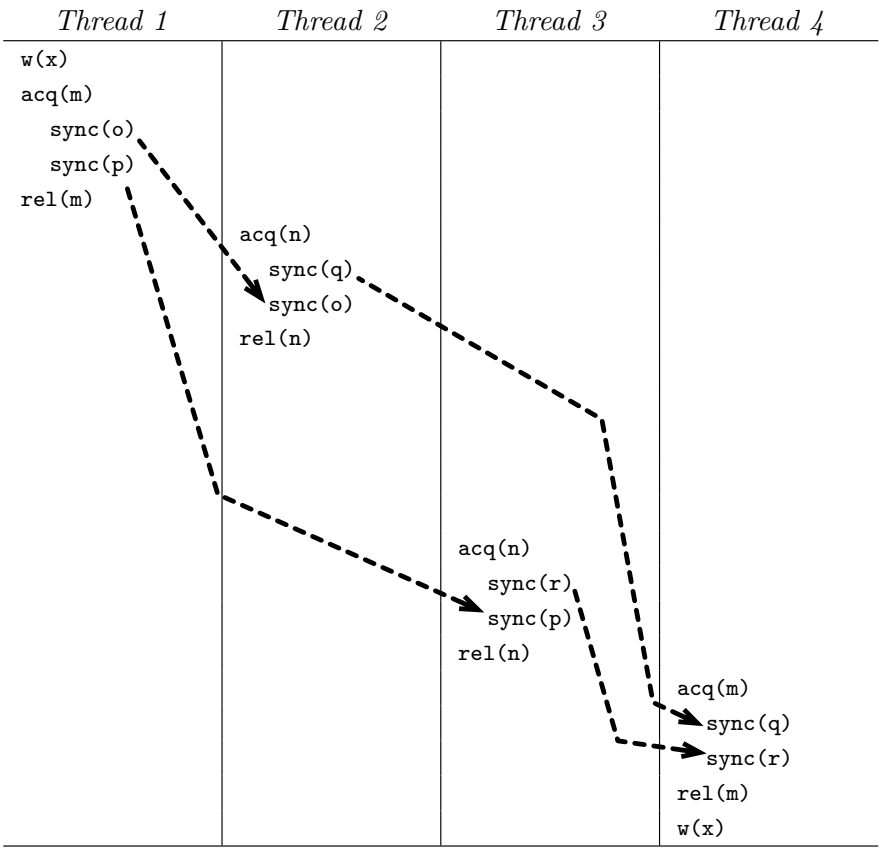
There is no predictable race between the two writes to `x` in the above example: any correct reordering of the execution will have the three `sync` sequences and the critical sections over lock `m` ordered in the way they were observed, resulting in an ordering of the two writes. Interestingly, the empty critical section over `l` by Thread 3 is necessary, or there would be a predictable race. The CP definition captures this reasoning accurately. Rule (b) is essential in establishing that the two critical sections over lock `l` are CP-ordered: because of rule (b), the end of the critical section over `l` in Thread 2 is CP-ordered relative to the (empty) critical section over `l` in Thread 3.

Since CP composes with HB to yield CP, the `sync(n)` event in Thread 2 is CP-ordered with `acq(l)` in Thread 3, thus triggering rule (b).

12.2 Example Highlighting Third Rule in CP Definition

For another interesting example, consider Figure 12.2. There is no predictable

Figure 12.2: Trace with no predictable HB-race on `x`.



race on `x` in this example, but establishing this fact requires case-based reasoning involving both the hard ordering constraints induced by `sync` sequences and the semantics and identity of locks (*e.g.* the fact that the critical sections on `n` cannot overlap). CP

avoids such reasoning but gives an accurate result. The two critical sections on `n` are not CP-ordered, and also do not necessarily occur in the order shown in a correctly reordered execution. The two critical sections on `m`, however, are CP-ordered and also necessarily in the order observed. In this example, we see the interesting aspects of Rule (c) of the CP definition: even though the HB order between the critical sections on `n` is what enables the CP order between the critical section on `m`, the former does not get upgraded to a CP order.

12.3 Example Highlighting Interconnectedness

Figure 12.3: No predictable race between the two writes to `x`.

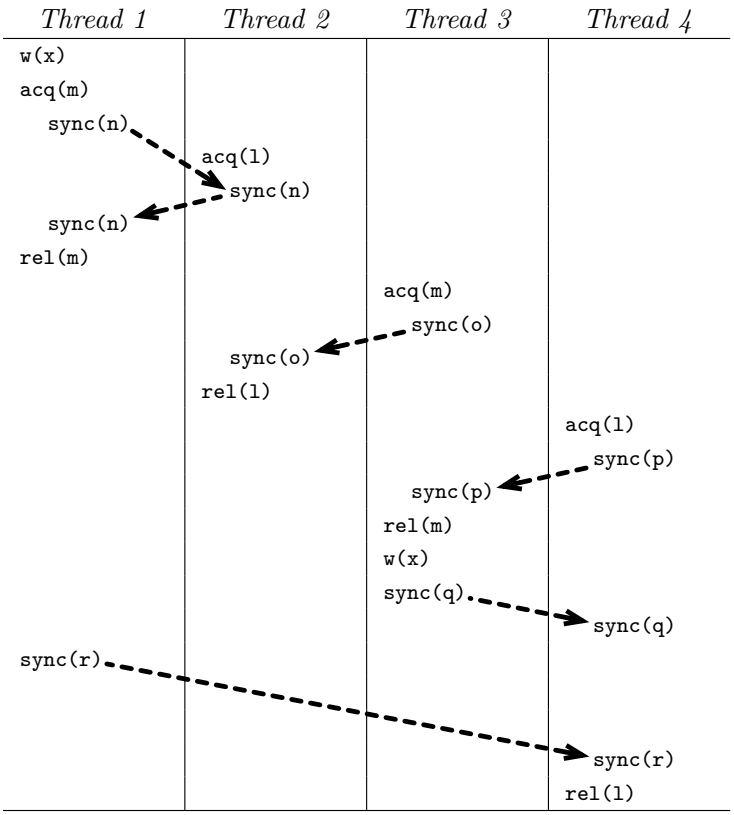


Figure 12.3 presents another hard-to-reason-about example. This execution does not have a predictable race on variable `x`. Nevertheless, the reasoning required to establish this fact can be quite complex. Removing events can easily result in a racy execution. For instance, removing the `sync(r)` edge allows a race by moving the entire set of operations by Thread 3 and Thread 4 before those of Thread 1 and Thread 2. This example highlights how potential relations must be able to deal with complicated schedules involving several threads and many dependencies.

12.4 Example Highlighting Complex Reorderings

Detecting predictable races can often require complex reorderings of events. The value of a polynomial but sound predictive race detector is that it avoids exploring all such reorderings. Consider the case of Figure 12.4. There is a CP-race between the

Figure 12.4: Exposing the HB race on `x` (execution on the left) requires a complex reordering of events (shown on the right).

Thread 1	*Thread 2*
`acq(m)`	
`  sync(o)`	
`  w(x)`	
`  acq(n)`	
`  rel(n)`	
`rel(m)`	
	`acq(n)`
	`  acq(m)`
	`  rel(m)`
	`  w(x)`
	`  sync(o)`
	`rel(n)`

Thread 1	*Thread 2*
	`acq(n)`
	`  acq(m)`
	`  rel(m)`
`acq(m)`	
`  sync(o)`	
`  w(x)`	
	`  w(x)`
	`  sync(o)`
	`rel(n)`
`  acq(n)`	
`  rel(n)`	
`rel(m)`	

two writes to variable `x`. There is also a predictable race. It is not possible to expose this, however, without thread scheduling that breaks up the synchronized block on `n` by Thread 2 and the synchronized block on `m` by Thread 1, as shown on the right part of the

figure. CP does not need to reproduce the schedule in order to warn of a possible race. We previously attempted to develop relations and proof strategies based on swapping entire synchronized blocks. This example highlights the pitfalls of such an approach.

12.5 CP and Deadlocks

CP often avoids complex nested lock reasoning by not distinguishing between a predictable race and a deadlock. The soundness theorem (Appendix B) has an interesting form: a CP-race is a sound indication of either a race or a deadlock in a correctly reordered execution. The deadlock is immediately apparent: there is a cycle in the lock-blocking graph. To see this consider the example of Figure 12.5. There is a CP-race

Figure 12.5: The observed execution (left) has a CP-race between the two writes to x. There is no predictable race, however! Instead, it is easy to reorder events to expose a deadlock (see right).

Thread 1	*Thread 2*
acq(m)	
acq(l)	
rel(l)	
w(x)	
rel(m)	
	acq(l)
	acq(m)
	rel(m)
	w(x)
	rel(l)

Thread 1	*Thread 2*
acq(m)	
	acq(l)
acq(l)	
	acq(m)

between the accesses to variable x in this example. Yet there is no predictable HB race: no correctly reordered execution can have the two write events without synchronization between them. The CP soundness theorem states that this is only possible when a reordering can expose a deadlock due to threads acquiring locks in a way that introduces a cycle in the acquisition dependencies.

The prior examples offer the reader a glimpse of the complexities of defining CP (and proving its soundness). The difficulty of reasoning about event order highlights the challenge of defining a relation that weakens the observed ordering much more than happens-before without resulting in false positives. There is a tradeoff between the complexity of a relation (and analyses based off of that relation) and the ability of that relation to predict additional races. A relatively simple race analysis has to conservatively assume ordering every time events *may* be ordered. Rule (c) of the CP definition plays this role but it was still difficult to prove that it is conservative enough. The ultimate conservative ordering is of course HB: all critical sections are assumed to always be precisely in the order they were observed.

Part V

Embracer

Chapter 13

Introduction

Precise race detectors are important tools for developing reliable multithreaded programs while avoiding the costs associated with false alarms. In Part IV, we presented a relation that enables precise race prediction. However, this relation is non-trivial to implement within a dynamic race detector. Part V, like Part IV, is focused on extending traditional dynamic race detection to also support *race prediction*. In Part V, we define an imprecise, online algorithm (EMBRACER) based on the must-before relation. Even when the observed trace is race-free, EMBRACER predicts if a race may occur on another, similar trace from the same program. This algorithm, like CP, is based on identifying situations where two synchronized blocks could be reordered. For example, EMBRACER successfully detects the predictable race in Trace A of Figure 10.1. However, the algorithm presented in this chapter has the potential to introduce false positives. The implementation for EMBRACER involves a variety of novel dynamic analysis techniques, such as adjustable vector clocks, and provides a significant increase in the ability to detect race conditions.

As a first step, we describe a *must-before* relation. This relation is an unsound approximation of the CP relation that is less restrictive (in terms of release-acquire edges) than the traditional happens-before relation, but easier to implement than the

CP relation. Whereas the happens-before relation includes edges between every two synchronized blocks that acquire the same lock, the must-before relation includes edges if the two synchronized blocks contain clashing memory accesses.

The happens-before relation is traditionally represented using vector clocks. For the the must-before relation, we develop representation techniques that are a bit more involved, since they require adding additional release-acquire edges "after the fact," when the race predictor is deep in the execution of the second synchronized block and detects a clash with an earlier block.

Chapter 14

Must-Before Relation

The happens-before relation totally orders synchronized blocks of the same lock via release-acquire edges. As we have seen in Part IV, this ordering is often too strict, since two synchronized blocks may be able to commute and still preserve behavior. In order to predict races that do not manifest in the current trace, this chapter introduces a new *must-before* relation that potentially contains less release-acquire edges than either the happens-before relation or the CP relation. As mentioned above, this relation can produce false positives. In particular, the tricky trace from Figure 12.2 contains an MB race on `x` because there is no MB edge between the synchronized blocks on `m`.[1]

Definition 8 (Must-Before Relation)**.** The must-before relation MB^α for a trace α is the smallest binary relation on operations in α that is:

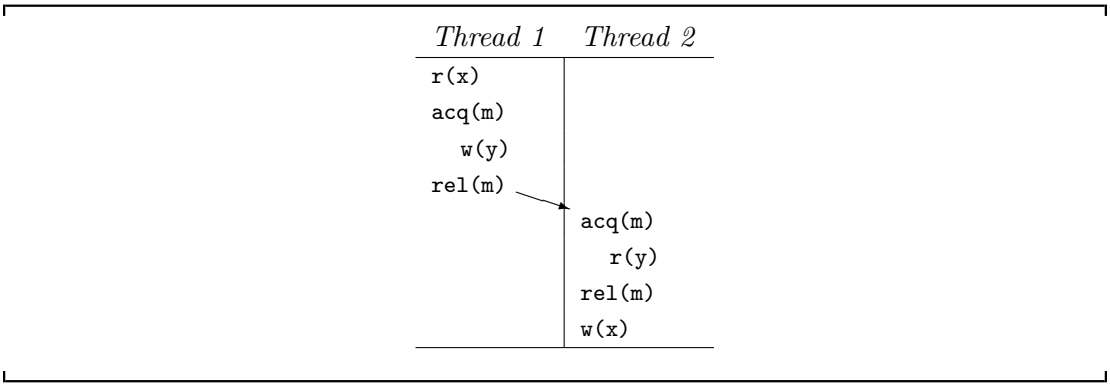
1. transitively closed,
2. includes PO^α and CO^α,
3. if $acq^k(t, m) <^\alpha_{\text{MB}} rel^h(u, m)$ where $k \neq h$
   then $rel^k(t, m) <^\alpha_{\text{MB}} acq^h(u, m)$

[1] In contrast, this edge exists in the CP relation because of the way CP is closed under composition with HB.

Note that although this relation contains less than or equal to the release-acquire edges of the CP relation, there are additional edges present between clashing accesses. This presence of additional edges simplifies the implementation of the algorithm, but means that a race detector based on the must-before relation may miss races identified by a standard happens-before detector. In practice, we integrate a standard happens-before detector into the implementation so as to report a superset of races. See Chapter 16 for more information.

Intuitively, condition 3 says that if the start $acq^k(t, m)$ of one synchronized block must occur before the end $rel^h(u, m)$ of a later synchronized block, then the end of the first block must occur before the start of the second block. To illustrate this situation, consider the following trace (Figure 14.1), where a clash on `y` between the two synchronized blocks orders Thread 1's acquire before Thread 2's release by transitivity, thereby inducing an MB edge between Thread 1's release and Thread 2's acquire.

Figure 14.1: A trace with a release-acquire MB edge



In contrast, in Figure 10.1(A) we do not include an MB edge between the release and acquire operations on `this` because there is no other MB relationship between the two synchronized blocks.

Note that we cannot determine if two synchronized blocks are ordered by the MB relation until we see the end of the second synchronized block, and so the MB

relation can only detect race conditions in closed traces (with no open synchronized blocks). A trace α has a *must-before race condition* if α is closed and there exists two clashing operations $a, b \in \alpha$ such that:

$$\nexists c \in \alpha.\ a <^{\alpha}_{\mathrm{MB}} c \text{ and } c <^{\alpha}_{\mathrm{PO}} b$$

In this situation we write $(a, b) \in \mathrm{MBR}^{\alpha}$.

Chapter 15

Must-Before Race Prediction

The central challenge in implementing a must-before race detector is adding edges, relating *past* operations, to the MB relation that is already encoded in the vector clocks. To illustrate this problem, consider again the trace in Figure 14.1, where a clash on `y` between the two synchronized blocks necessitates an additional MB edge between the first release and the second acquire: we may only add this release-acquire edge upon the *second* release. The traditional vector clock representation does not support adding past edges.

15.1 Adjustable Vector Clocks

Instead, we extend the vector clock representation by keeping a collection of extra release-acquire edges $\mathcal{E} : (K \times K)^*$. If we have $(k_1, k_2) \in \mathcal{E}$, then there is an "extra" edge to consider, from k_1 to k_2. This edge is useful only if $k_1 \not\sqsubseteq k_2$, since otherwise the edge is already encoded in k_1 and k_2.

Every use of a vector clock k needs to be "adjusted" according to these extra edges, via the function $adjust_{\mathcal{E}}(k)$, which returns the smallest vector clock k', such that

1. $k \sqsubseteq k'$; and

2. if $(r, a) \in \mathcal{E}$ and $a \sqsubseteq k'$ then $r \sqsubseteq k'$.

Intuitively, given an operation with vector clock k, if k has already been adjusted to k', *and* if $\mathcal{E}$ contains a must-before edge from r (release) to a (acquire) *and* there is a must-before edge from a to k' (*i.e.*, $a \sqsubseteq k'$), then by transitivity there is also a must-before edge from r to k' (*i.e.*, $r \sqsubseteq k'$).

We define the extended ordering relation $\sqsubseteq_{\mathcal{E}}$ that appropriately adjusts for these extra edges:

$$k_1 \sqsubseteq_{\mathcal{E}} k_2 \quad \text{if} \quad k_1 \sqsubseteq \mathit{adjust}_{\mathcal{E}}(k_2)$$

This adjustable vector clock representation of the must-before relation combines the efficiency and compactness of the vector clock representation with the flexibility of being able to add extra edges to the past.

15.2 Analysis State

We now present EMB RACER (a must-before race predictor), our algorithm for performing race prediction founded on the must-before relation. EMBRACER is expressed as an online algorithm based on an analysis state $\sigma = (\mathcal{C}, \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{H}, \mathcal{A}, \mathcal{B})$; we describe each of the state components below.

- $\mathcal{C} : \mathit{Tid} \to K$ records the vector clock of the current operation by each thread.
- $\mathcal{W} : \mathit{Var} \to K$ records the vector clock of the last write of each shared variable.
- $\mathcal{R} : \mathit{Var} \to K$ records the vector clock of the last read of each shared variable.
- $\mathcal{E} : (K \times K)^*$ records the extra edges connecting synchronized blocks in our MB relation.
- $\mathcal{H} : \mathit{Lock} \to (K \times K)^*$ records for each lock the history of acquire and release operations for each synchronized block.

Figure 15.1: Must-Before Race Detection Algorithm

[MB READ]

$$\frac{\begin{array}{rcl}\mathcal{B}' & = & \mathcal{B} \cup \{(\mathcal{W}_x, \mathcal{C}_t) \mid \mathcal{W}_x \not\sqsubseteq \mathcal{C}_t\} \\ \mathcal{C}' & = & \mathcal{C}[t := inc_t(\mathcal{C}_t \sqcup \mathcal{W}_x)] \\ \mathcal{R}' & = & \mathcal{R}[x := \mathcal{R}_x \sqcup \mathcal{C}'_t]\end{array}}{(\mathcal{C}, \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{H}, \mathcal{A}, \mathcal{B}) \Rightarrow^{r(t,x,v)} (\mathcal{C}', \mathcal{W}, \mathcal{R}', \mathcal{E}, \mathcal{H}, \mathcal{A}, \mathcal{B}')}$$

[MB WRITE]

$$\frac{\begin{array}{rcl}\mathcal{B}' & = & \mathcal{B} \cup \{\langle \mathcal{W}_x, \mathcal{C}_t\rangle \mid \mathcal{W}_x \not\sqsubseteq \mathcal{C}_t\} \cup \{\langle \mathcal{R}_x, \mathcal{C}_t\rangle \mid \mathcal{R}_x \not\sqsubseteq \mathcal{C}_t\} \\ \mathcal{C}' & = & \mathcal{C}[t := inc_t(\mathcal{C}_t \sqcup \mathcal{W}_x \sqcup \mathcal{R}_x)] \\ \mathcal{W}' & = & \mathcal{W}[x := \mathcal{C}'_t]\end{array}}{(\mathcal{C}, \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{H}, \mathcal{A}, \mathcal{B}) \Rightarrow^{w(t,x,v)} (\mathcal{C}', \mathcal{W}', \mathcal{R}, \mathcal{E}, \mathcal{H}, \mathcal{A}, \mathcal{B}')}$$

[MB ACQUIRE]

$$\frac{\begin{array}{rcl}\mathcal{C}' & = & \mathcal{C}[t := inc_t(\mathcal{C}_t)] \\ \mathcal{A}' & = & \mathcal{A}[m := \mathcal{C}_t]\end{array}}{(\mathcal{C}, \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{H}, \mathcal{A}, \mathcal{B}) \Rightarrow^{acq(t,m)} (\mathcal{C}', \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{H}, \mathcal{A}', \mathcal{B})}$$

[MB RELEASE]

$$\frac{\begin{array}{rcl}\mathcal{A}' & = & \mathcal{A}[m := \bot] \\ \mathcal{H}' & = & \mathcal{H}[m := \mathcal{H}_m \cup \{\langle \mathcal{A}_m, \mathcal{C}_t\rangle\}] \\ \mathcal{E}' & = & \mathcal{E} \cup \{\langle r, \mathcal{A}_m\rangle \mid \langle a, r\rangle \in \mathcal{H}_m \wedge a \sqsubseteq_{\mathcal{E}} \mathcal{C}_t\}\end{array}}{(\mathcal{C}, \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{H}, \mathcal{A}, \mathcal{B}) \Rightarrow^{rel(t,m)} (\mathcal{C}, \mathcal{W}, \mathcal{R}, \mathcal{E}', \mathcal{H}', \mathcal{A}', \mathcal{B})}$$

- $\mathcal{A} : \mathit{Lock} \to K_\bot$ records for each lock the vector clock of the open acquire, or $\bot$ if there is none.

- $\mathcal{B} : (K \times K)^*$ records the vector clocks for each potentially bad (or racy) clashing access pair. Since we determine must-before races only on closed traces (with no open synchronized blocks), we store clashing access pairs in $\mathcal{B}$ and check if they are actually races later, when we have a closed trace.

15.3 Analysis Rules

For each operation a in the observed trace, the must-before analysis state σ is updated via the transition relation $\sigma \Rightarrow^a \sigma'$, as described via the transition rules in Figure 15.1.

- [MB READ] records in $\mathcal{B}$ one potentially racy clashing access pair, containing the last write $\mathcal{W}_x$ to this variable and the current clock $\mathcal{C}_t$. Here, $\mathcal{C}$ is a map, $\mathcal{C}_t$ is an abbreviation for $\mathcal{C}(t)$, and $\mathcal{C}[t := k]$ denotes a map that is identical to $\mathcal{C}$ except that it matches t to k. Changes to the instrumentation state are expressed as functional updates for clarity in the analysis rules, but are implemented as in-place updates in our implementation. Pairs connected by an MB edge are elided for efficiency if $\mathcal{W}_x \sqsubseteq \mathcal{C}_t$. The $\mathcal{C}_t$ component is updated to reflect communication-order by joining the current vector clock with the last write. The $\mathcal{R}_x$ component is updated with the current vector clock.

- The [MB WRITE] rule records in $\mathcal{B}$ two potentially racy clashing access pairs from the last write and the last read to this variable, unless these clashing accesses are clearly race-free. The $\mathcal{C}_t$ component is updated to reflect communication-order by joining the current vector clock with the vector clock of the last write and last read, and incremented to order this write operation after previous writes and reads.

- The [MB ACQUIRE] rule records when the synchronized block is opened by storing the current vector clock in $\mathcal{A}_m$. The current vector clock is then incremented to order $\mathcal{A}_m$ before subsequent events.

- [MB RELEASE] records the synchronized block's open and close time in the history component $\mathcal{H}_m$.

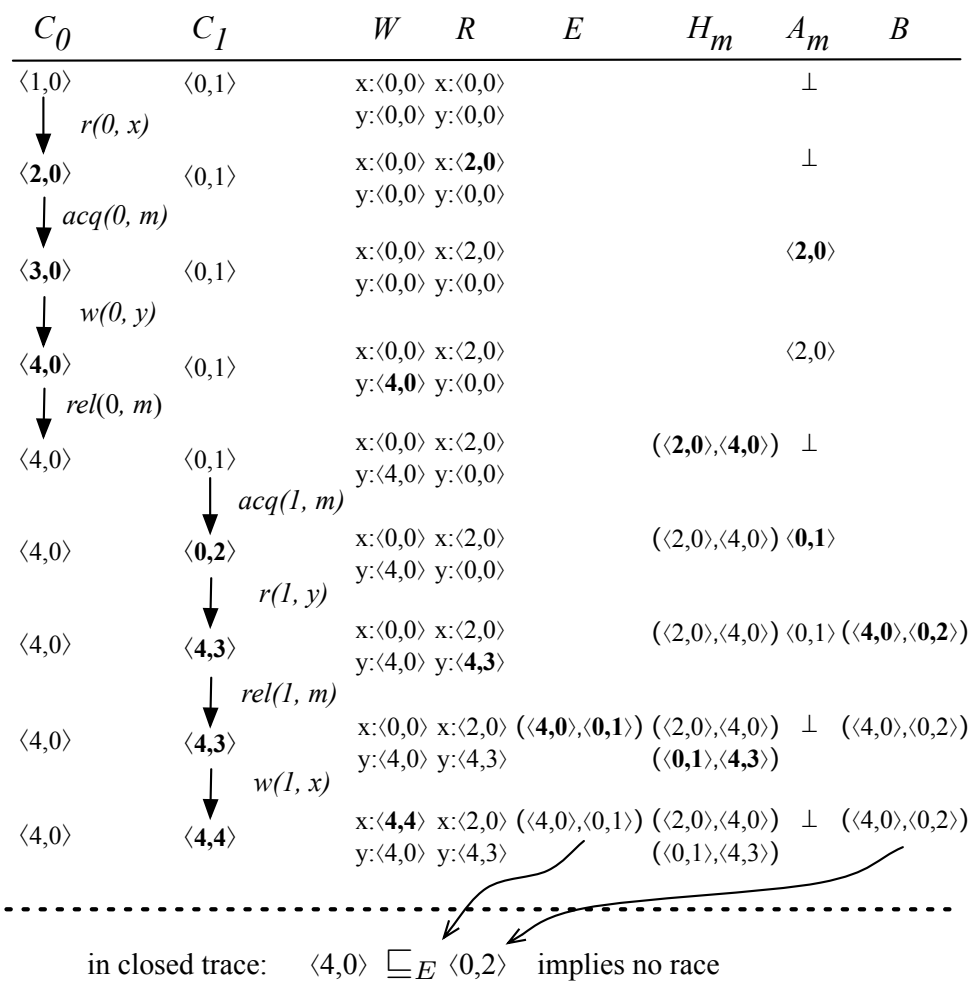
The final antecedent of this rule applies Condition 3 of Definition 8 to add extra release-acquire edges to $\mathcal{E}$. We consider all synchronized blocks (a, r) in $\mathcal{H}_m$, and if the acquire operation (at a) has a must-before edge to the current release operation (at $\mathcal{C}_t$), then we add an extra MB edge from the release of that block (at r) to the start of the current synchronized block (at $\mathcal{A}_m$). Although operationally awkward, $\mathcal{H}$ and $\mathcal{E}$ are necessitated by the fact that an MB release-acquire edge can only be inferred after the second synchronized block finishes its release operation.

We say that a state σ is *closed* if $\forall t \in \mathit{Tid}.\ \mathcal{A}_t = \bot$ (that is, if the trace is closed). When the analysis reaches a closed state, it processes all clashing access pairs in $\mathcal{B}$ to determine if any reflect a real race after adjusting for the extra edges in $\mathcal{E}$. Thus, we consider a state $\sigma = (\mathcal{C}, \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{H}, \mathcal{A}, \mathcal{B})$ to be *racy* if there exists $(a, b) \in \mathcal{B}$ such that $a \not\sqsubseteq_{\mathcal{E}} b$. In other words, a pair of clashing operations in $\mathcal{B}$ represents a reportable race if they are concurrent, even after adjusting for the release-acquire edges in $\mathcal{E}$.

Analysis Example. We now illustrate our analysis on the example trace of Figure 14.1. This trace has clashing accesses to both `x` and `y`, and the analysis needs to determine whether these clashing accesses are concurrent. Figure 15.2 shows the relevant portions of the analysis state, and how these state components are updated as each operation in the observed trace is processed.

The first three operations update R_x with the read time of `x`, A_m with the acquire time of lock `m`, and W_y with the write time for `y`, respectively. The operation $\mathit{rel}(0, m)$ updates H_m with a pair containing the acquire and release times for this synchronized block. The operation $r(1, y)$ appears concurrent with the previous write, so the corresponding vector clocks are added to the bad (or clashing) set $\mathcal{B}$, and $\mathcal{C}_1$ is updated to record the write-read CO edge on `y`. At $\mathit{rel}(1, m)$, the analysis detects that this synchronized block by Thread 1 does not commute with the earlier block, and so it adds the "extra" release-acquire edge separating these two blocks to $\mathcal{E}$.

Figure 15.2: Must-Before Vector Clock Example



Once the trace is closed, the analysis inspects the bad set $\mathcal{B}$ to determine if it reflects a real race by checking if $\langle 4,0\rangle \sqsubseteq_{\mathcal{E}} \langle 0,2\rangle$, or equivalently, if $\langle 4,0\rangle \sqsubseteq \mathit{adjust}_{\mathcal{E}}(\langle 0,2\rangle)$. The extra edge $(\langle 4,0\rangle, \langle 0,1\rangle)$ implies that $\mathit{adjust}_{\mathcal{E}}(\langle 0,2\rangle) = \langle 4,2\rangle$, and so the analysis concludes that $\langle 4,0\rangle \sqsubseteq_{\mathcal{E}} \langle 0,2\rangle$ and there is no race on `y`.

Chapter 16

Implementation

We implemented the EMBRACER dynamic race detector as part of the ROADRUNNER dynamic analysis framework, discussed in Chapter 9. The implementation closely follows the analysis. For each variable or array element x, we maintain a shadow object that records vector clocks $\mathcal{W}_x$ and $\mathcal{R}_x$. For each thread t, we maintain a thread state object that records $\mathcal{C}_t$. For each lock m, we maintain a shadow object that records $\mathcal{A}_m$ and $\mathcal{H}_m$. We represent $\mathcal{E}, \mathcal{H}$, and $\mathcal{B}$ (all of type $(K \times K)^*$) using a class that contains two equi-length arrays of vector clocks. We also store additional state information to support error reporting. Specifically, we keep pointers to the shadow state and thread state objects for each access in $\mathcal{B}$, along with information about the specific accesses.

16.1 Optimizations

We implemented several key optimizations to improve performance. A release-acquire edge (r, a) is elided from $\mathcal{E}$ if $r \sqsubseteq a$. We index $\mathcal{H}_m$ by thread identifier to add only a single release-acquire edge from a particular thread. Lastly, we implement a fast path by storing the thread identifier for the previous write of a variable, and bypass checking for a race on read operations if the thread currently reading wrote the value

being read.

16.2 Additional Operations

Figure 16.1 describes how our analysis handles additional operations: fork, join, wait, notify and volatile memory accesses. To support notify operations, we extend the analysis state with an additional component that stores for each lock the vector clock of the last notify:

$$\mathcal{N} : Lock \to K$$

- [MB VOLATILE READ] and [MB VOLATILE WRITE] are similar to the basic MB rules for reading and writing normal variables. Since volatile operations are synchronization operations, we do not add edges to $\mathcal{B}$.

- [MB FORK] reflects that a fork operation must occur before the first operation by the forked thread. Similarly, [MB JOIN] reflects that the last operation by a thread must occur before that thread is joined on.

- Our instrumentation framework provides separate prewait and postwait events that bracket each wait operation of a target program. [MB PRE WAIT] corresponds to the lock release executed directly before a thread goes to wait. In contrast, [MB POST WAIT] is executed when a thread finishes waiting and corresponds to a lock acquire, coupled with a join on $\mathcal{N}$.

- [MB NOTIFY] updates $\mathcal{N}$ and $\mathcal{C}$ components to impose a total order on the notify operations on each lock m.

Figure 16.1: Additional Operations

[MB VOLATILE READ]

$$\frac{\begin{array}{rcl} \mathcal{C}' & = & \mathcal{C}[t := inc_t(\mathcal{C}_t \sqcup \mathcal{W}_x)] \\ \mathcal{R}' & = & \mathcal{R}[x := \mathcal{R}_x \sqcup \mathcal{C}'_t] \end{array}}{\sigma \Rightarrow^{vr(t,x,v)} (\mathcal{C}', \mathcal{W}, \mathcal{R}', \mathcal{E}, \mathcal{H}, \mathcal{A}, \mathcal{B}, \mathcal{N})}$$

[MB VOLATILE WRITE]

$$\frac{\begin{array}{rcl} \mathcal{C}' & = & \mathcal{C}[t := inc_t(\mathcal{C}_t \sqcup \mathcal{W}_x \sqcup \mathcal{R}_x)] \\ \mathcal{W}' & = & \mathcal{W}[x := \mathcal{C}'_t] \end{array}}{\sigma \Rightarrow^{vw(t,x,v)} (\mathcal{C}', \mathcal{W}', \mathcal{R}, \mathcal{E}, \mathcal{H}, \mathcal{A}, \mathcal{B}, \mathcal{N})}$$

[MB FORK]

$$\frac{\mathcal{C}' = \mathcal{C}[t := inc_t(\mathcal{C}_t), u := inc_u(\mathcal{C}_u \sqcup \mathcal{C}_t)]}{\sigma \Rightarrow^{fork(t,u)} (\mathcal{C}', \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{H}, \mathcal{A}, \mathcal{B}, \mathcal{N})}$$

[MB JOIN]

$$\frac{\mathcal{C}' = \mathcal{C}[u := inc_u(\mathcal{C}_u), t := inc_t(\mathcal{C}_t \sqcup \mathcal{C}_u)]}{\sigma \Rightarrow^{join(t,u)} (\mathcal{C}', \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{H}, \mathcal{A}, \mathcal{B}, \mathcal{N})}$$

[MB PRE WAIT]

$$\frac{\text{identical to [MB RELEASE]}}{\sigma \Rightarrow^{prewait(t,m)} (\mathcal{C}, \mathcal{W}, \mathcal{R}, \mathcal{E}', \mathcal{H}', \mathcal{A}', \mathcal{B}, \mathcal{N})}$$

[MB POST WAIT]

$$\frac{\begin{array}{rcl} \mathcal{C}' & = & inc_t(\mathcal{C}_t \sqcup \mathcal{N}_m) \\ \mathcal{A}' & = & \mathcal{A}[m := \mathcal{C}_t] \end{array}}{\sigma \Rightarrow^{postwait(t,m)} (\mathcal{C}', \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{H}, \mathcal{A}', \mathcal{B}, \mathcal{N})}$$

[MB NOTIFY]

$$\frac{\begin{array}{rcl} \mathcal{C}' & = & \mathcal{C}[t := inc_t(\mathcal{C}_t \sqcup \mathcal{N}_m)] \\ \mathcal{N}' & = & \mathcal{N}[m := \mathcal{C}'_t] \end{array}}{\sigma \Rightarrow^{notify(t,m)} (\mathcal{C}', \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{H}, \mathcal{A}, \mathcal{B}, \mathcal{N}')}$$

16.3 Embedded Happens-Before Race Detection

In the EMBRACER algorithm presented thus far, the inclusion of CO edges may cause the algorithm to miss races identified by a traditional happens-before race detector. Specifically, the inclusion of CO edges enforces a sequentially consistent memory model, even for racy accesses, whereas the Java Memory Model does not enforce an ordering constraint between racy variable accesses.[1] The trace in Figure 16.2 illustrates this scenario. Under the algorithm presented so far, this trace has only one race on `y`; under a relaxed memory model, it has races on both `x` and `y`.

Prior precise dynamic race detectors (*e.g.* FASTTRACK [62]) exploit the relaxed memory model to detect more races. To address this disparity, the implementation of EMBRACER concurrently performs race prediction and traditional happens-before-based race detection for a relaxed memory model. In Chapter 17, the races found by EMBRACER are a superset of the races found by FASTTRACK. Incorporating relaxed memory model reasoning directly into the must-before relation is quite challenging, due to the subtle interactions between a relaxed memory model and race prediction, and remains a topic for future work.

Figure 16.2: Under sequentially consistent memory models, this trace has only one race on y; under the Java Memory Model, it has races on both x and y.

Thread 1	*Thread 2*
`w(x)`	
`w(y)`	
	`r(y)`
	`r(x)`

[1] See Section 3.2 for more information about memory models.

Chapter 17

Evaluation

We evaluated the performance and prediction capability of EMBRACER on a collection of multithreaded Java benchmarks ranging in size from 1,000 to 844,000 lines of code. These benchmarks include: `colt`, a library for high performance computing [30]; `raja`, a raytracer program [68]; `mtrt`, a raytracer program from the SPEC JVM98 benchmark suite [144]; and several benchmarks (`crypt`, `lufact`, `moldyn`, `montecarlo`, `raytracer`, `series`, `sor`, `sparse`) from the Java Grande set [81] (configured with four threads and the base data set).

We also performed experiments on three large reactive benchmarks: `Jigsaw`, W3C's web server [154], coupled with a stress test harness; `FtpServer`, a high-performance FTP server implementation from The Apache Foundation [150], coupled with a JMeter workload [151]; and several Eclipse benchmarks based on the Eclipse software development kit (version 3.4.0 [153]). The Eclipse benchmarks were automated using `swtbot`, a UI automation library for SWT and Eclipse [152]. For comparison purposes, each Eclipse benchmark extends the previous one with a new operation:

Startup: Launch Eclipse and immediately shut down.

NewProj: Launch Eclipse, create a new Java project, shut down.

Programs	Size (LOC)	Thread Count	Race Conditions Detected: Average Per Run: FastTrack	Average Per Run: Embracer	Average Per Run: Percent	Total Over 10 Runs: FastTrack	Total Over 10 Runs: Embracer	Total Over 10 Runs: Percent
PolarCoord	44	3	0.1	1.0	1000%	1	1	100%
montecarlo	3,669	4	1.0	1.0	100%	1	1	100%
mtrt	11,317	5	1.0	1.0	100%	1	1	100%
raytracer	1,970	4	1.0	1.0	100%	1	1	100%
Eclipse:	844,000	21						
Startup	-	-	(61.7)	(72.3)	117%	(68)	(81)	119%
NewProj	-	-	(63.9)	(74.6)	117%	(70)	(83)	119%
Build	-	-	72.3	86.9	120%	83	102	123%
Jigsaw	49,000	77	21.8	26.0	120%	33	35	106%
FtpServer	39,000	11	20.3	27.7	136%	27	35	130%
Total	945,000		117.5	144.6	**123%**	147	176	**120%**

Table 17.1: Races Detected By Embracer and FastTrack. Parenthesized numbers are not included in totals.

Build: Launch Eclipse, create a new Java project, build a project containing 50,000 lines of code, and shut down.

As a measure of complexity, more than 4000 classes were loaded and instrumented when executing the Eclipse-Build benchmark. It is unusual within the research community to run dynamic analysis tools on benchmarks of this size and complexity.

Excluding the Java standard libraries, all classes loaded by benchmark programs were instrumented. In all experiments, we used a fine granularity for array accesses, with a distinct shadow object for each index in an array. Like FastTrack, our tool includes special processing for the barrier operations in these benchmarks and does not report race conditions inside barrier code. We ran these experiments on a machine with 12 GB memory and two cores clocked at 2.8 GHz, running Mac OS X 10.6.1 with Java HotSpot 64-Bit Server VM 1.6.0.

17.1 Coverage and prediction

Since the central contribution of EMBRACER is its increased coverage, we compare its coverage with the FASTTRACK race detector [62], which is based on the happens-before relation. We ran each benchmark 10 times, simultaneously applying both EMBRACER and FASTTRACK to each trace to provide an objective comparison over precisely the same set of traces. Table 17.1 presents (in the "Average Per Run" columns) the average number of races detected in each run. For clarity, we omit benchmarks from the table that had 0 races for both tools. To avoid double-counting, the totals exclude the Eclipse Startup and NewProj benchmarks, since they are essentially subsumed by the Build benchmark. These results show that, although EMBRACER is about twice as slow as DJIT$^+$, EMBRACER detects 20-23% more races than FASTTRACK for reactive programs.

Table 17.1 also reports the lines of code for each benchmark. For the larger benchmarks, these numbers were calculated by totaling the number of lines of code in each class that was actually instrumented when running the benchmark. The total number of source lines of code for Eclipse alone is several million. Of the omitted benchmarks, `lufact`, `moldyn`, `series`, `sor`, and `sparse` were all 4 threads and between 1,000 and 1,600 LOC; `colt` has 11 threads and 111,000 LOC; and `crypt` has 7 threads and 1,200 LOC.

17.1.1 Relation to CP

Examining the results for CP race detection (Appendix A) and EMBRACER, we find that race prediction has the most dramatic results when run on reactive programs, including GUIs and webservers. We believe this is because in our benchmark set, we have already found most of the (few) races for the compute-bound benchmarks with a happens-before analysis, while the reactive benchmark set is much more nondetermin-

Program	Size (LOC)	Base Time (sec)	Time Slowdown				
				Relaxed MM		SCMM	
			EMPTY	FASTTRACK	DJIT+	HBTOOL	EMBRACER
colt	25,644	16.1	0.8	0.9	1.0	0.9	0.9
crypt	1,241	0.4	8.5	19.7	65.7	74.0	77.9
lufact	1,627	47.1	0.3	0.3	0.4	0.5	0.6
moldyn	1,402	8.6	1.4	2.0	2.9	7.4	8.3
montecarlo	3,669	1.9	5.0	11.1	36.4	38.1	41.0
mtrt	11,317	0.7	10.2	11.1	13.8	14.7	14.9
raja	12,828	0.5	7.4	8.1	8.1	9.2	9.3
raytracer	1,970	1.2	5.8	14.0	23.1	87.9	94.9
series	967	2.0	1.6	1.6	1.7	1.7	1.7
sor	1005	0.8	6.4	7.3	13.0	14.7	15.6
sparse	868	0.5	7.8	22.9	49.0	92.3	111.5
Average			5.0	9.0	19.5	31.0	34.2

Table 17.2: Performance of EMBRACER; 10% over HBTOOL.

istic and involves many more threads and many more potential interleavings.

After 10 runs, race prediction (either with CP or EMBRACER) only provides a bonus for the reactive programs: `Jigsaw`, `FtpServer`, and `Eclipse`. Unfortunately, the trace-based CP-analysis does not scale well to the Eclipse benchmarks, which produce multi-gigabyte sized traces. The results for CP-based prediction and EMBRACER for `Jigsaw` and `FtpServer` (Table 17.1 and A.1) are very similar. For `FtpServer`, CP finds 7 additional races and EMBRACER finds 8 additional races, as compared with 10 runs of a happens-before race detector. For `Jigsaw`, CP finds 3 additional races and EMBRACER finds 2 additional races, as compared with 10 runs of a happens-before race detector.

17.2 Performance

To evaluate the performance of EMBRACER, we compare it to three other precise dynamic race detectors, all implemented on top of the same framework: FASTTRACK [62], DJIT$^+$ [118], and HBTOOL. HBTOOL is an optimized happens-before race detector with communication-order edges added; we include this in the table to to give a better sense of the overhead entailed for tracking communication-order edges Table 17.2 compares the slowdown of each compute-bound benchmark under each of these four race detectors, plus under EMPTY, which measures the overhead of our instrumentation framework. For the results reported in this table, we ran EMBRACER without concurrent happens-before race detection. However, we found that the embedded happens-before race detector inside of EMBRACER does not significantly change the performance. Each slowdown is the average of 10 runs; there was very little deviation across runs.

The results show that EMBRACER's performance is competitive with HBTOOL, despite its need to use significantly more complex data structures (such as $\mathcal{H}, \mathcal{E}, \mathcal{B}$); the average slowdown of EMBRACER is 34 versus 31 for HBTOOL: a 10% overhead. The memory overhead of EMBRACER compared with either HBTOOL or DJIT$^+$ is also about 10%.

Both DJIT$^+$ and FASTTRACK are noticeably faster because they use additional optimizations to the HB relation that are valid for a relaxed memory model. We believe that a tool implementation involving an extension of the MB relation targeted towards relaxed memory models could also implement such optimizations and so offer comparable performance to FASTTRACK, but exploring these issues remains a topic for future work. To put these performance numbers in perspective, the race detector Helgrind from the Valgrind suite [110] can have slowdowns of 300x [155].

Part VI

SideTrack

Chapter 18

Introduction

Atomicity guarantees that a program's behaviour can be understood as if each atomic block executes without interference from other threads. In Part VI, we focus on dynamically detecting atomicity violations in traditional multithreaded programs. Even if the observed trace is serializable, our online analysis can still infer that the original program can generate other feasible traces that are not serializable. This work was originally presented at the 2009 Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD) [171].

To illustrate this idea, consider the trace in Figure 18.1, where Thread 1 is executing an atomic block containing two synchronized blocks, Thread 2 is executing a single synchronized block, and the vertical ordering of the statements of the two threads reflects their relative execution order. Here, `beg(a)` and `end(a)` demarcate the begin and end of an atomic block (labelled `a`) within the execution trace. We use "..." to indicate a (possibly empty) sequence of operations when we do not care about the actual specific sequence. In future figures, `fork(t)` denotes forking thread `t`.

Clearly, this trace is serial and hence trivially serializable, and so a precise atomicity checker such as VELODROME [66] would not detect any errors. A careful analysis of this trace, however, shows that the synchronized block on `l` by Thread 2 *could*

Figure 18.1: Observed Serial Trace

Thread 1	*Thread 2*
`beg(a)`	
`acq(l)`	
`...`	
`rel(l)`	
`acq(l)`	
`...`	
`rel(l)`	
`end(a)`	
	`acq(l)`
	`...`
	`rel(l)`

have been scheduled in between the two synchronized blocks of Thread 1; the original source program that generated the trace of Figure 18.1 is also capable of generating the non-serializable trace shown in Figure 18.2.

Figure 18.2: Feasible Non-Serializable Trace

Thread 1	*Thread 2*
`beg(a)`	
`acq(l)`	
`...`	
`rel(l)`	
	`acq(l)`
	`...`
	`rel(l)`
`acq(l)`	
`...`	
`rel(l)`	
`end(a)`	

Technically, the synchronized block of Thread 2 could diverge in this alternate trace. The analysis presented in this part assumes that all synchronized blocks terminate, and that the program is free of race conditions. This latter assumption can be discharged by concurrently running an efficient precise race detector such as FastTrack [62] or Goldilocks [51].

We would like to catch the violation of Figure 18.2 without having to observe

it directly. Whenever the same lock is acquired twice within a transaction, there is a *vulnerable window* between the two acquires where a *culprit* acquire by another thread could cause an atomicity violation.

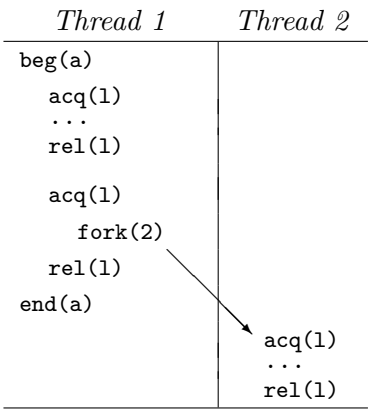
SideTrack Part VI introduces SideTrack, an online predictive atomicity analysis. By observing the serial trace of Figure 18.1, our dynamic analysis can detect that the source program contains an atomicity violation, even though that violation does not manifest itself in the current trace. Thus, our analysis generalizes from the observed trace to detect errors that *are guaranteed* to occur on another feasible trace. SideTrack may sometimes produce false positives, *e.g.* in the case of infinite loops, killing threads, or shutting down the Java Virtual Machine. However, in most cases the errors found will be real.

Our analysis detects three kinds of errors – BEFORE-ERRORs, IN-ERRORs, and AFTER-ERRORs – depending on where the culprit acquire occurs in relation to the vulnerable window, as shown in Figure 18.4 (a), (b), and (c), respectively. In these figures, the vulnerable window is depicted as a circled V. Prior precise dynamic atomicity tools such as Velodrome only detect IN-ERRORs; SideTrack introduces the additional ability to detect BEFORE-ERRORs and AFTER-ERRORs.

Although the examples in Figure 18.4 are rather straightforward, in practice generalizing from the observed trace without introducing false positives may be rather involved. To illustrate some of the issues, consider an alternate trace shown in Figure 18.3, where the second synchronized block of Thread 1 forks Thread 2. In this situation, the synchronized block of Thread 2 cannot be scheduled between the two synchronized blocks of Thread 1 (this dependency is depicted with an arrow in the figure).

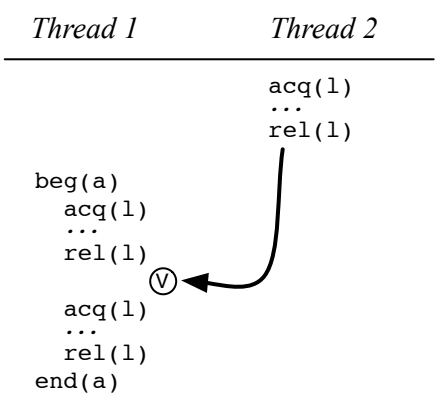
Despite the similarities between Figures 18.1 and 18.3, the first trace reflects an atomicity error in the source program, whereas the second trace does not. SideTrack

Figure 18.3: Observed Serial Trace with Fork

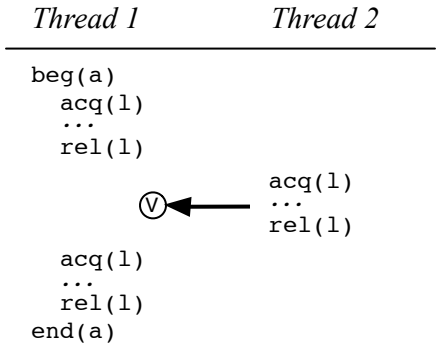


needs to perform a careful analysis of the happens-before relation of the observed trace to detect situations where certain synchronized blocks could have been scheduled before other synchronized blocks. SideTrack uses the standard technique of vector clocks to provide a compact and efficient representation of this happens-before relation. SideTrack tracks the relative timing of synchronization operations and flags an error if an operation by another thread is concurrent with one of the operations flanking a vulnerable window.

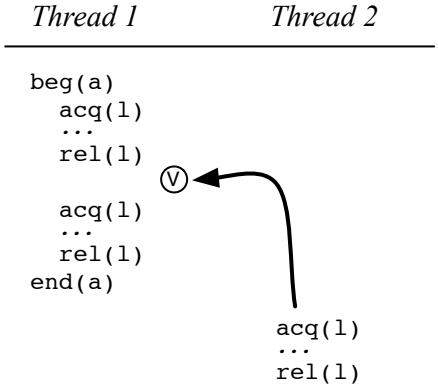
Figure 18.4: Three Kinds of Atomicity Violations



(a) BEFORE-ERRORS



(b) IN-ERRORS



(c) AFTER-ERRORS

Chapter 19

Foundations

SideTrack builds off of the foundations presented in Part III in a variety of ways. In this chapter, we make the additional formal background required by SideTrack clear.

For SideTrack, we extend the set of explicitly listed operations that thread t can perform to also include:

- $begin^l(t)$ and $end^l(t)$, which demarcate each `atomic` [66] (or `deterministic` [126]) block labelled l;
- $fork(t, u)$, which forks a new thread u;
- $join(t, u)$, which blocks until thread u terminates.

Traces fulfill expected constraints when forking and joining; for example, no operations by a thread u occur in a trace prior to the forking of thread u.

We also need to extend the notion of *conflict* to the following additional condition:

- **Fork-join conflict**: one operation is either $fork(t, u)$ or $join(t, u)$ and the other operation is by thread u.

A *transaction* in a trace α is the sequence of operations executed by a thread t starting with a $begin^l(t)$ operation and containing all t operations up to and including a matching $end^l(t)$ operation. To simplify some aspects of the formal presentation, we assume $begin^l(t)$ and $end^l(t)$ operations are appropriately matched and are not nested (although our implementation does support nested atomic specifications).

In a *serial* trace, all operations from each transaction are grouped together and not interleaved with the operations of any other transaction. A trace is serializable if it is equivalent to a serial trace.

If two operations in a trace have a fork-join or program order conflict, then we say they have an *enables conflict*. The *enables relation* $\prec^\alpha$ is the smallest transitively-closed relation on operations in α such that if operation a occurs before b in α and a has an enables conflict with b, then a *enables* b.

A lock operation a in α is *concurrent* with a later lock operation b by a different thread unless there exists an operation c between a and b such that $a <^\alpha_{HB} c$ and $c \prec^\alpha b$. This is a tricky definition and is important for the later discussion, so it is worth developing the intuition behind it. In essence, in a hypothetical alternate trace where b occurs before a, c would also need to occur before a. However, once the happens-before edge between c and a is reversed, the thread that executed c may take an alternate path of execution. Since c enables b, this means that b may not occur if c occurs earlier.

In any good definition of concurrent, two concurrent operations may execute in either order. Therefore, to identify concurrent operations we need to ensure that a dependency does not exist between them. The happens-before relation identifies dependencies but is too restrictive; every pair of lock operations on the same lock are related by happens-before. The enables relation is too weak; the lock operations on m by different threads in Figure 20.2(a) are not related by enables, but it may not be possible to execute them in a different order (Figure 20.2(b)).

Chapter 20

Analysis

Our analysis detects situations where a feasible but not observed atomicity violation can be predicted dynamically. Whenever the same lock is acquired twice within a transaction, there is a *vulnerable window* between the two acquires where an acquire by another thread could cause an atomicity violation. Our analysis keeps track of these vulnerable windows, and flags an error if it detects that a feasible trace exists that exploits a vulnerable window to cause an atomicity violation. We refer to the external acquire that could occur in the vulnerable window as a culprit acquire.

Note that when predicting feasible traces dynamically, it is not possible to predict the entire trace. For example, we can predict that a lock acquire could have occurred between the two previous acquires, but the trace after that point may not be predictable based on the information obtained from the first trace. Instead, our analysis can only predict that *there exists* some alternate trace with an atomicity violation, where the alternate trace shares a prefix with the observed trace and executes certain synchronized blocks in a certain order.

Because of these inherent limitations in our ability to generalize from the observed trace, some traces that appear at first glance to have a predictable atomicity violation turn out to be more complicated upon closer inspection. For example, con-

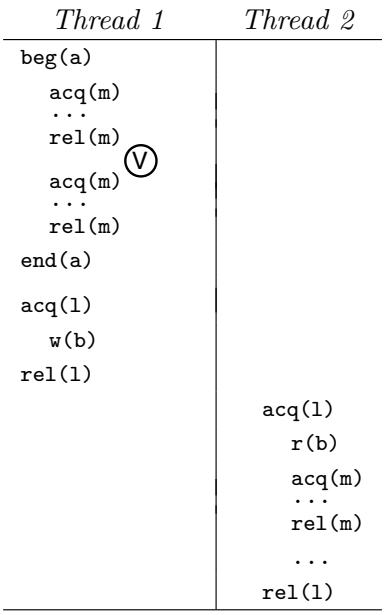
sider the program in Figure 20.1, which could produce the trace in Figure 20.2(a). In this trace, it appears the synchronized block of Thread 2 could execute inside the vulnerable window of Thread 1. However, if the read of `b` by Thread 2 executes before the write to `b` by Thread 1, we can no longer predict the subsequent execution of Thread 2, and in particular cannot guarantee that Thread 2 will still synchronize on the lock `m` (Figure 20.2(b)). The notion of concurrency defined in Section 19 captures these constraints.

Figure 20.1: Example Program `STExample`

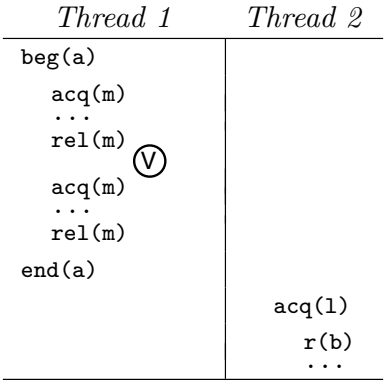
```
1  class STExample {
2    Mutex m, l;
3    boolean b;
4    static STExample e = new STExample();
5
6    atomic void vulnerableFunction() {
7      synchronized(m) { ... }
8      synchronized(m) { ... }
9    }
10
11   void setb() {
12     synchronized(l) { b := true; }
13   }
14
15   atomic void checkb() {
16     synchronized(l) {
17       if (b) { synchronized(m) { ... } }
18       else { ... }
19     }
20   }
21
22   public static void main(String[] args){
23     fork { e.vulnerableFunction(); e.setb(); }
24     fork { e.checkb(); }
25   }
26 }
```

As stated in the introduction, our analysis detects three different kinds of errors (BEFORE-ERRORs, AFTER-ERRORs, and IN-ERRORs), depending on where the culprit acquire occurs in relation to the vulnerable window.

Figure 20.2: A non-vulnerable atomic block due to the happens-before edge on the lock `l`, which protects the variable `b`.



(a) Thread 2 synchronizes on lock `m`.



(b) Thread 2 may not synchronize on lock `m`, due to an alternate control flow.

In a BEFORE-ERROR, the culprit acquire occurred before the vulnerable window, as illustrated in Figure 18.4(a). To detect BEFORE-ERRORs, the analysis records the vector clock of the most recent acquire of each lock. When a lock is first acquired in an atomic block, the analysis checks if that acquire is concurrent with the previous acquire of that lock, via a vector clock comparison. If the two acquires are concurrent, then that lock is recorded as being *potentially interfering* in that atomic block; if the atomic block subsequently re-acquires that lock, then a BEFORE-ERROR atomicity violation is reported.

Note that it is not enough to keep track of the most recent *release* of each lock. It is possible for a lock release by Thread 1 to be concurrent with an acquire of the same lock by Thread 2, while the corresponding acquire by Thread 1 enables the acquire by Thread 2. In the most extreme case, Thread 1 could fork Thread 2 while holding the the lock.

In an IN-ERROR, the culprit acquire occurs in the vulnerable window, as in Figure 18.4(b). Other dynamic atomicity analysis tools that do not generalize to additional traces typically catch IN-ERRORs; an IN-ERROR represents an actual atomicity violation in the observed trace. We catch IN-ERRORs inside the vulnerable window, when the program is about to execute the second acquire within the transaction. There is an atomicity violation if a thread is about to acquire the same lock twice within a transaction and discovers the last release of that lock is concurrent with the acquire about to happen.[1]

In an AFTER-ERROR, the culprit acquire occurs after the vulnerable window, as in Figure 18.4(c). We catch AFTER-ERRORs when the program is about to execute the culprit acquire. We keep track of the most recent vulnerable window for every lock. If the vector clock for an acquire by a different thread is not later than the vulnerable

[1]Concurrent operations are by different threads, so this would mean that another thread acquired the lock.

window, than that thread could have executed the acquire in the vulnerable window.

20.1 Analysis formalization

Based on these ideas, we now formally define our atomicity analysis as an online algorithm based on an analysis state $\omega = (C, V, A, R, H, I)$ where:

- $C : \mathit{Tid} \to K$ records the vector clock of the current operation by each thread;
- $V : \mathit{Lock} \to K$ records the vector clock of the most recent vulnerable window for each lock;
- $A : \mathit{Lock} \to K$ records the vector clock of the last acquire of each lock;
- $R : \mathit{Lock} \to K$ records the vector clock of the last release of each lock;
- $H : \mathit{Tid} \to 2^{\texttt{LOCKS}} \cup \{\texttt{NotInX}\}$ records the set of locks held within the current transaction by each thread, or `NotInX` if that thread is not currently within a transaction; and
- $I : \mathit{Tid} \to 2^{\texttt{LOCKS}}$ records the set of potentially interfering locks for each thread.

In the initial analysis state, all vector clocks are initialized to $\bot$, except each C_t starts at $\mathit{inc}_t(\bot)$ to reflect that the first steps by different threads are not ordered.

$$\begin{aligned} \omega_0 \quad = \quad & (\lambda t.\, \mathit{inc}_t(\bot), \\ & \lambda m.\, \bot, \\ & \lambda m.\, \bot, \\ & \lambda m.\, \bot, \\ & \lambda t.\, \texttt{NotInX}, \\ & \lambda t.\, \emptyset) \end{aligned}$$

Figure 20.3: SIDETRACK Atomicity Violation Detection Algorithm

[ST ENTER]

$$\frac{\begin{array}{rcl} H' & = & H[t := \emptyset] \\ I' & = & I[t := \emptyset] \end{array}}{(C, V, A, R, H, I) \Rightarrow^{begin^l(t)} (C, V, A, R, H', I')}$$

[ST EXIT]

$$\frac{H' = H[t := \texttt{NotInX}]}{(C, V, A, R, H, I) \Rightarrow^{end^l(t)} (C, V, A, R, H', I)}$$

[ST READ]

$$\frac{}{(C, V, A, R, H, I) \Rightarrow^{r(t,x,v)} (C, V, A, R, H, I)}$$

[ST WRITE]

$$\frac{}{(C, V, A, R, H, I) \Rightarrow^{w(t,x,v)} (C, V, A, R, H, I)}$$

[ST FORK]

$$\frac{C' = C[t := inc_t(C_t), u := C_t \sqcup C_u]}{(C, V, A, R, H, I) \Rightarrow^{fork(t,u)} (C', V, A, R, H, I)}$$

[ST JOIN]

$$\frac{C' = C[t := C_t \sqcup C_u, u := inc_u(C_u)]}{(C, V, A, R, H, I) \Rightarrow^{join(t,u)} (C', V, A, R, H, I)}$$

[ST FIRST ACQUIRE]

$$\frac{\begin{array}{rcl} H_t & \neq & \texttt{NotInX} \qquad m \notin H_t \\ \texttt{if } V_m & \not\sqsubseteq & C_t \texttt{ then AFTER-ERROR} \\ C' & = & C[t := C_t \sqcup R_m] \\ A' & = & A[m := C_t] \\ H' & = & H[t := H_t \cup \{m\}] \\ I' & = & (\text{if } A_m \sqsubseteq C_t \text{ then } I \text{ else } I[t := I_t \cup \{m\}]) \end{array}}{(C, V, A, R, H, I) \Rightarrow^{acq(t,m)} (C', V, A', R, H', I')}$$

[ST SECOND ACQUIRE]

$$\frac{\begin{array}{rcl} m & \in & H_t \\ \texttt{if } m & \in & I_t \texttt{ then BEFORE-ERROR} \\ \texttt{if } R_m & \not\sqsubseteq & C_t \texttt{ then IN-ERROR} \\ \texttt{if } V_m & \not\sqsubseteq & C_t \texttt{ then AFTER-ERROR} \\ C' & = & C[t := C_t \sqcup R_m] \\ V' & = & V[m := V_m \sqcup C_t] \\ A' & = & A[m := C_t] \end{array}}{(C, V, A, R, H, I) \Rightarrow^{acq(t,m)} (C', V', A', R, H, I)}$$

[ST OUTSIDE ACQUIRE]

$$\frac{\begin{array}{rcl} H_t & = & \texttt{NotInX} \\ \texttt{if } V_m & \not\sqsubseteq & C_t \texttt{ then AFTER-ERROR} \\ C' & = & C[t := C_t \sqcup R_m] \\ A' & = & A[m := C_t] \end{array}}{(C, V, A, R, H, I) \Rightarrow^{acq(t,m)} (C', V, A', R, H, I)}$$

[ST RELEASE]

$$\frac{\begin{array}{rcl} C' & = & C[t := inc_t(C_t)] \\ R' & = & R[m := C_t] \end{array}}{(C, V, A, R, H, I) \Rightarrow^{rel(t,m)} (C', V, A, R', H, I)}$$

Figure 20.3 shows how the analysis state is updated for each operation a of the target program.

The first rule [ST ENTER] for $begin^l(t)$ records that thread t is in a new transaction by switching H_t away from `NotInX` to $\emptyset$, and resets the set of interfering locks. The complementary rule for $end^l(t)$ records that t is no longer in a transaction. Here, H is a function, H_t abbreviates the function application $H(t)$, and $H[t := V]$ denotes the function that is identical to H except that it maps t to V. Changes to the instrumentation state are expressed as functional updates for clarity in the analysis rules, but are implemented as in-place updates in our implementation.

Read and write operations do not affect the analysis state. If required, race conditions can be detected by running SideTrack concurrently with a race detector such as FastTrack [62]. We do not yet consider reads and writes of volatile variables, which create nonblocking synchronization, although we believe we can adapt our analysis to handle these constructs.

We update vector clocks for fork and join operations to reflect the structure of the enables relation. The rule [ST FORK] for $fork(t, u)$ performs one "clock tick" for thread t and sets the vector clock associated with u to be greater than previous operations by thread t. The rule [ST JOIN] records that the last operation of the joined thread enables the join operation. The vector clock for the joined thread is incremented to preserve the invariant that the current vector clocks for each running thread are incomparable.

There are three analysis rules associated with an $acq(t, m)$ operation. All three rules share three common antecedents which:

1. Update the vector clock for t to reflect that the current time for t is later than

the previous release of m;

$$C' \quad = \quad C[t := C_t \sqcup R_m]$$

2. Update the last acquire for m appropriately; and

$$A' \quad = \quad A[m := C_t]$$

3. Report an AFTER-ERROR if the vulnerable window for m is not before the current clock for t:

$$\texttt{if } V_m \quad \not\sqsubseteq \quad C_t \texttt{ then AFTER-ERROR}$$

The rule [ST OUTSIDE ACQUIRE] applies to acquires that are not within a transaction, and simply performs the above three actions.

The rule [ST FIRST ACQUIRE] applies the first time a thread acquires a lock within a transaction. We add m to the set of locks H_t acquired within that transaction, and check if the previous acquire of m was concurrent. If so, we add m to the set I_t of potentially interfering locks for that thread.

The rule [ST SECOND ACQUIRE] applies the second time a thread acquires a lock within a transaction. We record the new vulnerable window for m in V_m. We flag a BEFORE-ERROR if there is a previous interfering lock that could have executed after the first acquire by t. We flag an IN-ERROR if the previous release of m is not before the current clock for t. All previous operations by t, including the first acquire of m by t, are before the current clock of t. Therefore, if $R_m \not\sqsubseteq C_t$, then another thread released (and previously acquired) m between the two acquires by t.

The rule [ST RELEASE] for $rel(t, m)$ updates the vector clock associated with the latest release for m. We also perform one clock tick for thread t.

20.2 Examples

Figure 20.4: Illustration of the Analysis State When Discovering an AFTER-ERROR

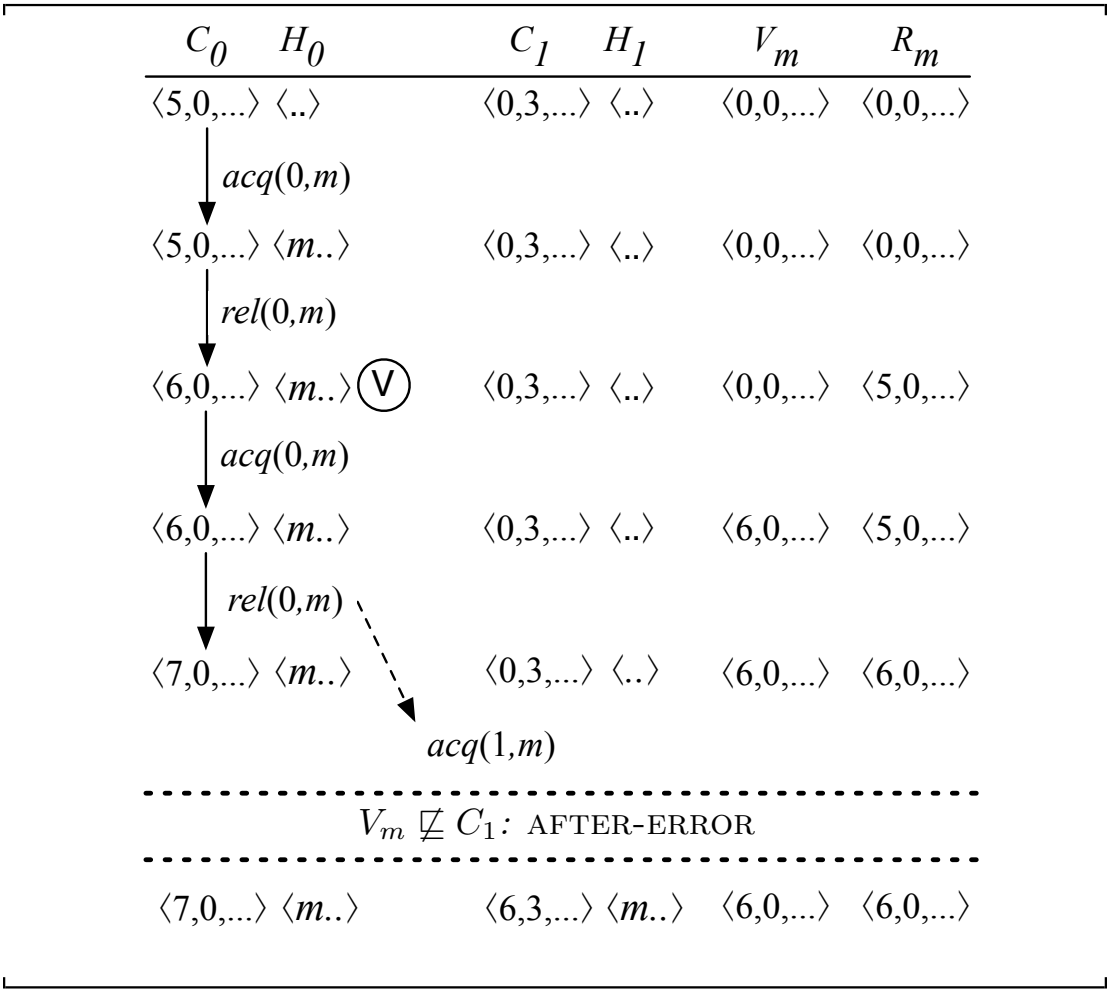
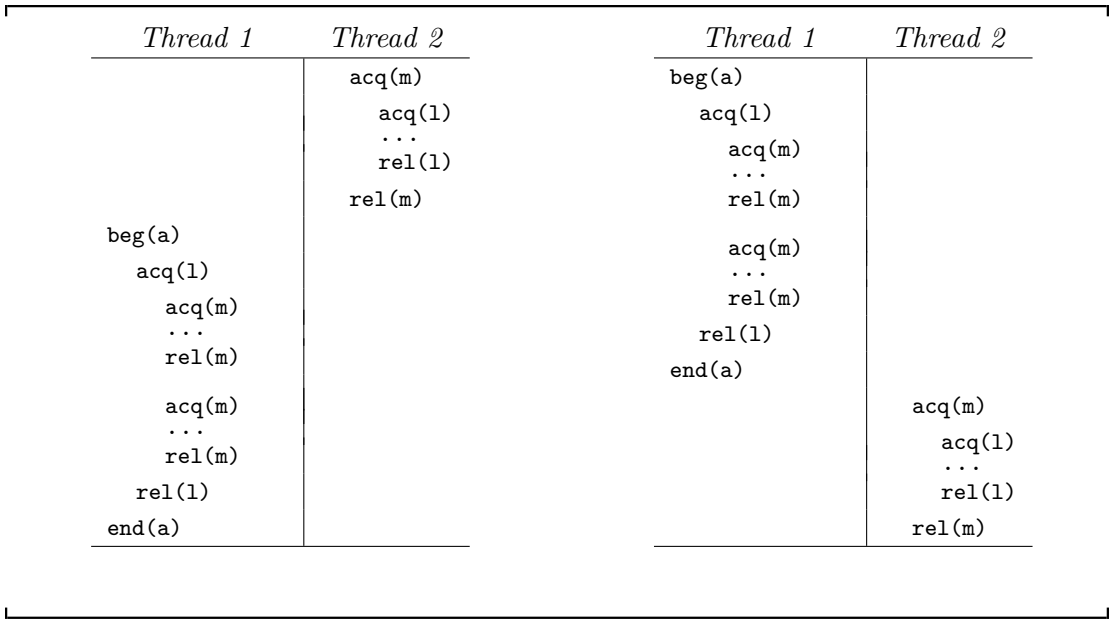


Figure 20.4 shows how vector clocks are updated on a sample trace fragment involving an AFTER-ERROR. The I and A parts of the analysis state play no part in discovery of AFTER-ERRORs, so they are omitted from the figure for simplicity. After the first release by thread 0, C_0 is incremented from $\langle 5, 0, \ldots \rangle$ to $\langle 6, 0, \ldots \rangle$ and R_m is set to $\langle 5, 0, \ldots \rangle$. When m is acquired a second time by thread 0, V_m is set to the current time for thread 0 ($\langle 6, 0, \ldots \rangle$). At the second release by thread 0, C_0 is incremented

again from $\langle 6, 0, \ldots \rangle$ to $\langle 7, 0, \ldots \rangle$ and R_m is updated to $\langle 6, 0, \ldots \rangle$. When Thread 1 goes to acquire m, an AFTER-ERROR is reported because $V_m \not\sqsubseteq C_1$. Once the AFTER-ERROR is reported, C_1 is joined with the time of the last release (R_m).

Figure 20.5: SIDETRACK flags the trace on the right with an atomicity violation, but not the trace on the left.



The example in Figure 20.5 illustrates that AFTER-ERRORs are more amenable to detection than BEFORE-ERRORs. SIDETRACK finds the AFTER-ERROR in the right trace because the acquires of `m` by the two threads are concurrent. In the left trace, the synchronization on `l` means that the acquires of `m` are *not* concurrent: the acquire of `l` by Thread 2 (and, by transitivity, the acquire of `m` by Thread 2) happens-before the acquire of `l` by Thread 1 and the acquire of `l` by Thread 1 enables the acquires of `m` by Thread 1. Therefore, a BEFORE-ERROR is not detected in this case.

Chapter 21

Implementation

We have developed a prototype implementation, called SideTrack, of our dynamic atomicity analysis. SideTrack is a component of RoadRunner, discussed in Chapter 9. By default, SideTrack checks for the specification that all methods are atomic. This lifts transactions to the method call level, and nested transactions represent nested method calls.

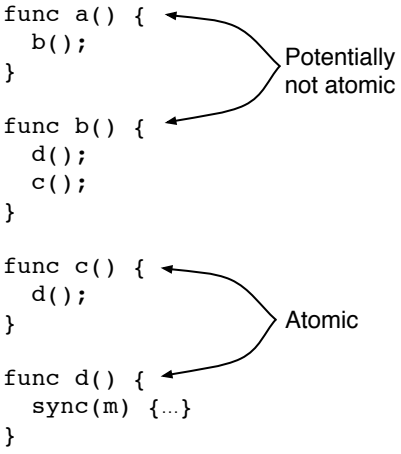
Each of the analysis rules in Figure 20.3 consists of one or more antecedents and a single consequent. Three rules are necessary to describe the event handler for acquires. When implementing these rules, it is useful to break down the antecedents into *conditionals*, *checks*, and *updates*. Conditionals indicate where the code must branch in an event handler. Checks indicate when the analysis should flag an error. Updates describe how the next analysis state is obtained.

For example, the rule [ST first acquire] has two branches with negated counterparts in different rules: $H_t \neq \texttt{NotInX}$ and $m \notin H_t$, and one internal branch: if $A_m \sqsubseteq C_t$.... The crucial check is $V_m \sqsubseteq C_t$. If this check is violated, SideTrack issues an after-error warning.

Blame assignment Our analysis as presented is relatively straightforward to implement, and successfully flags feasible atomicity violations. Accurate blame assignment requires significant extra work, especially since BEFORE-ERROR and AFTER-ERROR atomicity violations do not manifest in the observed trace.

In the presence of nested transactions, it is not sufficient to report the (inner) transaction where the second acquire of a vulnerable window occurs. To illustrate this difficulty, consider the program shown in Figure 21.1, where we assume that all methods should be atomic.

Figure 21.1: Program Example Where Precise Blame Assignment Is Necessary.



While methods `c` and `d` are atomic, `a` and `b` potentially have atomicity violations. When we encounter the second acquire in the vulnerable window inside of `b`, the call stack may look like `a:b:c:d`. However this call stack is not precise enough to tell us which method is non-atomic. In fact, we need the call stack for the first acquire of m inside the vulnerable window as well: `a:b:d`. We need to identify the common prefix in the two stacks, and report all methods in this prefix as non-atomic (here, `a` and `b`). We store a call stack for the first and second acquires for the last vulnerable window in the

instrumentation state associated with each lock, and call stacks for all locks acquired in the current transaction in the instrumentation state associated with each thread. With a suitable immutable linked-list representation, recording such a stack is a constant-time operation.

Tool complementation We can combine SideTrack with Velodrome [66], a sound and complete dynamic atomicity analysis, to increase coverage. Velodrome finds more atomicity violations from a particular trace than SideTrack, since Velodrome also looks for atomicity violations involving accesses to shared variables as well as locks. However, SideTrack finds feasible errors that Velodrome misses, since the latter does not generalize its analysis to other feasible traces. Thus the two tools are compatible and complementary, just like the combination of Embracer and traditional happens-before race detection. Since RoadRunner supports tool composition, it is straightforward to run these two together.

Chapter 22

Evaluation

We present encouraging experimental results for SideTrack. We ran these experiments on a machine with 3 GB memory, 2.16 GHz dual-core CPU, running Mac OS X 10.5.6, and with Java HotSpot 64-Bit Server JVM 1.6.0.

Our benchmark set includes: `elevator`, a real-time discrete event simulator [161]; `colt`, a library for high performance scientific computing [30]; `jbb`, the SPEC JBB2000 simulator for business middleware [143]; `hedc`, a warehouse web-crawler for astrophysics data [161]; `barrier`, a barrier synchronization performance benchmark [81]; `philo`, a dining philosophers application [51]; `tsp`, a solver for the traveling salesman problem [161]; and `sync`, a synchronization performance benchmark [81]. Benchmarks from Java Grande [81] were configured to use 4 threads and the base data set. This set totals over 200K lines of code. Excluding the Java standard libraries, all classes loaded by benchmark programs were instrumented.

To check for atomicity violations, we assumed that *all methods should be atomic*. In practice this assumption works fairly well; previous experiments have validated that atomicity is a fundamental design principle for concurrency [61, 98].

For each benchmark, Table 22.1 reports both the total number of methods with atomicity violations (Column 2) as well as the error count for each type of atom-

Programs	Methods with Errors	Types of Errors			Predicted Errors		
		BEFORE-ERRORs	IN-ERRORs	AFTER-ERRORs	BEFORE-ERRORs	AFTER-ERRORs	Total
elevator	5	3	1	5	2	4	4
colt	9	4	7	9	2	2	2
jbb	10	7	5	10	5	5	5
hedc	4	1	4	4	0	0	0
barrier	1	1	1	1	0	0	0
philo	1	1	1	1	0	0	0
tsp	4	4	4	4	0	0	0
sync	4	4	4	4	0	0	0
Total	38	25	27	38	9	11	11

Table 22.1: Atomicity Errors Found by SideTrack.

icity violation (Columns 3 - 5). Note that there is significant overlap between the methods reported by each kind of violation. To further clarify the benefit of SideTrack's predictive analysis, Column 6 reports the number of BEFORE-ERRORs that are not IN-ERRORs (and so not detected by earlier precise tools), and Column 7 reports on AFTER-ERRORs that are not IN-ERRORs. Finally, the last column reports on errors that are either BEFORE-ERRORs or AFTER-ERRORs, but not IN-ERRORs, and so most clearly summarizes the improvement achieved via predictive analysis. SideTrack's predictive ability catches an additional 11 errors, in addition to the 27 IN-ERRORs: an improvement of roughly 40%. Interestingly, all 11 of these additional violations were found by the *after* analysis, which suggests that the *after* analysis generalizes better than the *before* analysis.

To confirm that the predicted errors are not false positives, we investigated the error messages in the "Predicted Total" column by inspecting the program source

Table 22.2: SideTrack Benchmark Performance Results

Programs	Num Threads	loc	Base Runtime (Seconds)	Slowdown	
				Empty	SideTrack
colt	11	111,421	16.2	1.2	1.2
barrier	4	774	55.2	1.0	1.0
tsp	5	706	1.1	2.6	3.7
sync	4	650	68.8	0.8	0.9
crypt	7	1,241	0.6	3.9	4.3
moldyn	4	1,402	1.7	3.1	4.2
forkjoin	187	591	0.04	45.0	47.0
lufact	4	1,627	0.3	8.8	9.4
montecarlo	4	3,669	2.4	2.3	2.6
raytracer	4	1,970	1.5	5.4	13.9
series	4	967	2.9	1.5	1.8
sor	4	1,005	0.3	5.8	7.0

code. We found that it was easy to pinpoint the errors with the blame assignment information. All were judged to be vulnerable to atomicity violations. However, some of these violations may be benign.

Table 22.2 reports on the performance of our analysis. For each of the compute-bound benchmarks, we report on the running time of that benchmark (without any instrumentation), and on the slowdown incurred by RoadRunner when running with the Empty tool (which just measures the instrumentation overhead but performs no dynamic analysis); and the slowdown when run with SideTrack. We also measured performance for all other benchmarks in the Java Grande suite; SideTrack reported no errors (of any of the three types) for these additional benchmarks. The average slowdown for SideTrack was 8.1x, as compared with a slowdown for the Empty tool of 6.8x.

There are a couple outliers. The `forkjoin` benchmark taxes our instrumentation framework by forking and joining almost 200 threads, each of which has an associated instrumentation state. Additionally, the running time of this benchmark is so short that other effects (like printing) may dominate the slowdown. Without `forkjoin`, the slowdown for SideTrack is 4.5x and the slowdown for the Empty tool is 3.3x. The

13.9x slowdown for `raytracer` is a result of the large number of small method calls in this benchmark, each of which must be recorded for blame assignment.

The results show that SideTrack provides a significant improvement in performance of prior dynamic atomicity analyses, such as SingleTrack [126] (10.4x), Velodrome [66] (10.3x), and Atomizer [61]. This performance improvement is largely because SideTrack does not analyze memory reads and writes, and instead assumes that the target program is race-free and any inter-thread communication is mediated via synchronization idioms such as locks. If necessary, this race-free assumption can be verified by concurrently running an efficient race detector such as FastTrack [62], which would incur an additional performance overhead of 10x. We believe using separate analyses to verify race-freedom and atomicity offers benefits both in terms of performance and modularity of the analysis code.

Part VII

Tiddle

Chapter 23

Introduction

Part VII is focused on Tiddle: a language-agnostic domain-specific language (DSL) for describing trace behaviour. Tiddle can be used to generate deterministic tests for validating dynamic analysis tools. This work was originally presented at the 2009 International Workshop on Dynamic Analysis (WODA) [127].

As discussed in Part II, many dynamic analysis tools exist for discovering concurrency errors in programs. The tools are often intricate concurrent programs in their own right. Testing dynamic analyses involves writing many small programs that exhibit a fault (or do not exhibit a fault) to check if the analysis correctly runs. However, since these tests must be multithreaded, they may not be deterministic! One (inadequate) method of attempting to force a deterministic schedule is to pepper `yield()` or `sleep()` statements throughout the tests; this does not guarantee determinism, even if the statements are placed properly. The tests must also contain a lot of boilerplate code for setting up multiple threads. In short, they are both tedious and error-prone.

What is a better way of quickly testing dynamic analyses? It is our observation that small *traces* exhibiting certain faults are a natural way of describing test cases. Writing small traces on the whiteboard as an aid to discussion is natural. For example, the easiest way to explain a data race is to produce an example of one, and it is very easy

to do just that with Tiddle. This observation is supported by user studies; modeling failure traces is the only effective way to debug multithreaded programs [67].

We have created a trace description language, Tiddle, that captures basic concurrency notions in execution traces. Our trace language provides a simple abstraction of a multi-threaded program execution trace. We translate traces into *deterministic* Java source code for the purpose of testing dynamic analyses. This guarantees that the same execution trace results from every run of the program. We are able to achieve this determinism using a key assumption: that the dynamic analysis framework can optionally ignore specified method calls.

We have implemented a compiler, written in Haskell, which translates Tiddle trace statements into deterministic concurrent programs written in Java source code. The language and implementation is extensible, allowing customization to suit the needs of the user. We also use Tiddle to generate a variety of traces that exhibit the same behaviour as the source trace, so as to verify that a dynamic analysis behaves as expected on equivalent traces, or to generate nondeterministic small tests that exhibit a particular type of multithreading error.

We have used Tiddle as a development aide for dynamic analyses. As a concrete example, we developed SideTrack (Part VI) using Tiddle. During implementation, we discovered a bug in how SideTrack handled nested locking by a simple Tiddle trace. When we extended our analysis with another atomicity violation pattern, it was easy to test that the new pattern worked properly. Finally, when refactoring SideTrack's implementation, we used all the Tiddle traces as a regression test suite. In addition, we have used Tiddle to test the SingleTrack [126] deterministic parallelism checker. To date, 70 or so Tiddle programs have been written.

Chapter 24

Methodology

24.1 Tiddle Grammar

We describe the grammar of Tiddle. A BNF-style representation is given in Figure 24.1.

Figure 24.1: Tiddle Grammar

```
trace ::= trace op
        | op
op ::= rd   Tid Var (Val)
     | wr   Tid Var (Val)
     | acq  Tid Lock
     | rel  Tid Lock
     | fork Tid Tid
     | join Tid Tid
     | beg  Tid Label
     | end  Tid Label
Tid ::= Int
Var ::= String
Val ::= Int
Label ::= String
Lock :: = String
```

In Tiddle, a trace is a list of operations. Each operation belongs to a thread,

and the thread identifier is indicated by an integer in the first argument. Reads and writes specify a variable and optionally a value. If a read operation specifies a value, an `assert` statement is added to verify that the read returned the expected value; variables are initialized to 0 by default. Since the generated programs are deterministic, this is simply a check to make sure the trace is specified as desired. Acquires and releases specify a monitor lock. Forks and joins specify the thread identifier for the forked or joining thread. Begin and end operations demarcate atomic blocks, or transactions [61].

In Java, the reads and writes correspond to accesses to a static field. The acquires and releases correspond to a `synchronized` block. Forks and joins correspond to Java's `start()` and `join()` methods. Begin and end operations are Java blocks annotated as `atomic`. Alternatively, begin and end operations can denote method boundaries. This allows the generated programs to be checked with a general "all methods are atomic" specification. These generated Java programs represent simple, straightforward examples that embody the buggy pattern described by the corresponding Tiddle trace. The implementation could be easily extended to other languages that support concurrency and mutable fields (*e.g.* C++); this extension would involve some pretty printing changes in the module that translates Tiddle ASTs to Java ASTs.

24.2 Data Races

Tiddle operations support modeling data races (defined in Section 1.3.1). Our compiler generates working, complete Java code from a partial trace. There is no need to write out the full execution trace of a program; only the relevant lines of the trace need to be specified and other operations (*e.g.* forking all the threads involved) are added automatically. Here is a data race specified in two lines (`x` is initially `0`):

```
rd 1 x
wr 2 x 1
```

There is no explicit synchronization to prevent the trace from being reordered to:

```
wr 2 x 1
rd 1 x
```

so the final value of `x` – 0 or 1 – depends on the nondeterministic schedule of operations. Specifying this data race is only two lines of Tiddle code, but these two lines are translated to more than 50 lines of Java source code (Figure 24.2).

24.3 Atomicity Violations

Atomicity (discussed in Section 1.3.2) is a general concurrency specification that is focused on non-interference of code blocks. We can straightforwardly describe an atomicity violation in Tiddle:

```
beg 1 a
rd  1 x
wr  2 x
rd  1 x
end 1 a
```

The atomic block is indicated by the begin and end operations. For this trace to have the property of atomicity, it must be equivalent to a serial trace where the atomic block is executed without other threads interleaving. Here, the write to `x` in between two subsequent reads of `x` means that this trace is not equivalent to any serial execution of the atomic block. This trace compiles to about 70 lines of Java source code.

Note that the operations and their semantics reflect the kind of dynamic analyses we are interested in for this dissertation: detecting data races and atomicity violations. Tailoring the Tiddle language to operations for different dynamic analyses, such as detecting atomic-set-serializability violations [71] or new types of concurrency bugs should be straightforward.

Figure 24.2: Generated Java Code with Race

```
1  public class Test {
2    static int x = 0;
3    static CyclicBarrier cb = new CyclicBarrier(2);
4    static CyclicBarrier cc = new CyclicBarrier(2);
5    static int numThreads = 2;
6
7    static public void await(CyclicBarrier c)
8      throws BrokenBarrierException,
9                InterruptedException {
10     c.await();
11   }
12
13   public static void main(String[] args) {
14     final Thread t2 = new Thread() {
15       public void run() {
16         try {
17           int _z = 0; //for reads
18           await(cc);
19           await(cb);
20           await(cc);
21           x = 1;
22           await(cb);
23         } catch (InterruptedException e) {
24           e.printStackTrace();
25         } catch (BrokenBarrierException e) {
26           e.printStackTrace();
27         }
28       }
29     };
30     final Thread t1 = new Thread() {
31       public void run() {
32         try {
33           int _z = 0;
34           await(cc);
35           _z = x;
36           await(cb);
37           await(cc);
38           await(cb);
39         } catch (InterruptedException e) {
40           e.printStackTrace();
41         } catch (BrokenBarrierException e) {
42           e.printStackTrace();
43         }
44       }
45     };
46     t1.start();
47     t2.start();
48   }
49 }
```

24.4 Determinism and Synchronization

Race conditions and atomicity violations are nondeterministic by nature. A test program that has such errors is difficult to use, because the error may manifest rarely and only under specific conditions. Thus, test programs should be deterministic, even those with nondeterministic bugs.

We use barrier synchronization to implement determinism for otherwise nondeterministic test programs. The test program generated from the specification trace executes only one operation per barrier. All threads in the program move in lockstep from one barrier call (`await()`) to the next. Between each barrier call, precisely one thread executes an operation, while all other threads simply race to the next barrier and block.

We assume that dynamic analysis frameworks may elide certain method calls from being instrumented. We believe this is a reasonable assumption for any dynamic analysis framework, since deciding what to instrument depends on what analysis is being performed. With this capability to elide method calls, we ignore all calls to `await()` when analyzing the test program. The program continues to run deterministically; however, the analysis tool will not observe the barrier synchronization. If barrier calls were not elided from the analysis, the test program will still run deterministically. However, the analysis tool may now emit false positives (or negatives) because the tool reacts to the observed contention on the barrier.

Forks and joins present interesting synchronization issues, because these operations change the number of threads racing to a barrier. We use the `CyclicBarrier` class from the Java standard library, which allows recycling the barrier between waits, although a `BrokenBarrierException` gets thrown if the number of threads to trip a barrier is changed while some threads are still blocking on that barrier. We solve this problem by using two `CyclicBarrier`s, so that the second of the two barriers can be

reset to a new value while the other threads are all blocking on the first one.

Optionally, all barrier synchronization may be elided during code generation to produce *small nondeterministic* programs that exhibit a particular type of multi-threading error. This way, Tiddle can still be used for test generation in situations where ignoring `await()` calls is not possible. These could also be small test cases for testing static analysis tools. Alternatively, Tiddle can be configured to add `sleep()` or `yield()` statements to the (barrier-free) code, although the test programs will still be nondeterministic.

24.5 Code Generation

The compiler for the Tiddle language is written in Haskell [76]: a lazy, pure, strongly-typed functional language. Haskell has powerful constructs that make it easy to concisely define and manipulate abstract syntax trees (ASTs). We used Alex [101] and Happy [100] (Haskell versions of Lex and Yacc [95]) to generate a lexer and parser, respectively. The Haskell implementation generates a (custom) Java AST from the trace AST provided by the parser. Code generation is handled by a separate pretty printer [116] for the Java AST. Haskell is a terse language: our entire implementation totals about 300 lines of code. The functional style of Haskell make it a natural choice for AST generation and manipulation; we chose this language because of the combination of conciseness and expressivity, and because it was fun to use.

24.6 Equivalent Traces

In addition to generating the code for one particular trace, Tiddle can also generate all traces equivalent (Chapter 7) to a particular trace. We scan through the trace to find source operations that have only outgoing happens-before edges, and then recursively add all interleavings of operations respecting happens-before order. In essence,

this pulls out all equivalent traces by exploring different paths through the nodes in the happens-before graph for a trace, only adding an operation when all prior operations that happen-before it have been added to the trace. A series of equivalent traces can be compiled into one (long) Java program, enabling a single run of the analysis to confirm that all equivalent traces find the same errors (or lack thereof).

Part VIII

Future Work

Chapter 25

Future Work

In this dissertation, we have presented a variety of techniques related to dynamic prediction of concurrency errors. To enable precise dynamic prediction of data races, we introduced the CP relation. To further address dynamic predictive race detection, we then introduced the *must-before* relation and accompanying dynamic analysis tool (EMBRACER) that is not precise but enables online prediction. We also described SIDETRACK, a lightweight dynamic analysis tool that generalizes from an observed trace to predict additional atomicity violations. Lastly, we presented the Tiddle DSL and compiler for deterministic testing of dynamic analysis tools using trace snippets.

There are several clear areas for future improvement. We have yet to develop a fully online algorithm that works with relaxed memory models incorporating the ideas from the CP relation or EMBRACER. Such an algorithm should have a clear performance improvement over EMBRACER, as it could implement a variety of optimizations targeted towards relaxed memory models. We have not yet discovered a way to analyze for CP-races using techniques such as vector clocks (as is derived for the must-before relation in Chapter 15), nor have we discovered a full CP implementation that only does online reasoning (*i.e.* never needs to "look back" in the execution trace); these remain challenging questions for future work.

SideTrack could also be expanded to find new classes of atomicity violations beyond the specific locking patterns this tool now targets. It is possible that the CP relation could be adapted to perform atomicity detection, although we expect that such an implementation would be much more complex than SideTrack.

Another direction for future work is to better characterize the space of dynamic concurrency error prediction. There may be additional precise relations that are weaker than CP or otherwise provide an alternative to CP. Due to the many challenges we had developing CP (illustrated by the various tricky traces presented in Chapter 12), we envision the development of future relations to be a major undertaking.

An additional area for future work is extending the concurrency error prediction techniques explored in this dissertation to new classes of concurrency errors. There may be unique challenges to address when predicting deadlocks or determinism violations. We could also explore incorporating details gleaned from static analyses to improve the error detection capabilities of the techniques presented here.

A completely different direction is improving the error messages and other output produced by dynamic analysis tools. Little is known about how real developers use program analyses as part of their development workflow, or the most critical changes that would improve the usability of such tools. Perhaps static information or witness trace snippets could be incorporated into tool output to aid in the debugging process.

There are also areas of future work for Tiddle (Part VII). One potential extension would create small test programs to replicate faults, perhaps by recording buggy traces of large programs to instantiate smaller programs that exhibit the same multi-threaded bug. This is similar to record-and-replay, but replaying the problem instead of program: perhaps better suited for prototyping dynamic analyses. Another way to take Tiddle to the next level would be to develop it into a formal specification language for bugs such as data races and atomicity violations. This specification language may serve as an easier way of understanding safety properties through violation specifications.

Chapter 26

Conclusion

This dissertation has demonstrated that dynamic analysis can discover concurrency errors that do *not* manifest on the observed trace. We presented four tools to enable predictive dynamic analysis:

1. CP relation: A relation for predictive race detection.
2. Embracer: A predictive online race detector.
3. SideTrack: A predictive online atomicity analysis.
4. Tiddle: A domain specific language (DSL) for testing dynamic analyses.

This dissertation includes two main contributions to predictive race detection: the CP relation and the Embracer algorithm. The simplified tradeoffs of these two approaches are summarized in Figure 26.1. Causally-precedes (CP) is a weaker relation than the happens-before relation (upon which most precise dynamic race detectors are built), yet is strong enough to enable precise race detection. The examples described in Chapter 12 clarify why it is not easy to weaken HB without introducing false positives. A single CP-based race detection run discovers several new races, unexposed by 10 independent runs of plain HB race detection. However, an efficient online algorithm

	Post-Mortem	Online Algorithm
No False Positives	**CP relation**	*Future Work*
Maybe False Positives	*Prior Work*	**EMBRACER**

Figure 26.1: Simplified design space for predictive dynamic race detectors.

leveraging the CP relation remains a topic for future work. To address this gap, we presented an imprecise *must-before* relation to enable online prediction. This relation is leveraged by the EMBRACER algorithm to predict race conditions. EMBRACER detects 20-23% more races than a happens-before detector alone for the reactive programs analyzed in this dissertation.

This dissertation also addresses predictive atomicity violation detection. We showed how serializable traces can still reveal atomicity violations in the original program and presented SIDETRACK, a lightweight dynamic analysis tool that generalizes from an observed trace to predict atomicity violations that could occur on other feasible traces. SIDETRACK identifies a class of atomicity violations that result from a particular pattern: two acquires of the same lock inside an atomic block. Experimental results demonstrate that the predictive ability increases the number of atomicity violations detected by SIDETRACK by 40%.

Deterministic testing enables quick development of predictive dynamic analysis tools but concurrent programs are often nondeterministic. A dynamic analysis developer is often interested in the behaviour of their analysis on specific execution traces, such as the example traces from Chapter 12. We presented a methodology for deterministic testing of dynamic analysis tools using trace snippets. This methodology involves a language-agnostic DSL for describing trace behaviour, Tiddle, and an associated compiler to translate Tiddle traces into deterministic concurrent Java programs.

The four tools described in this dissertation represent an important first step for dynamic prediction of concurrency errors. There are many topics of future work, as described in Chapter 25, which will build off the research presented here. We look forward to watching this future unfold.

Part IX

Appendices

Appendix A

Experimental Results for CP

In this appendix, we reproduce preliminary experimental results for a CP-based race detector for reference. CP reasoning, based on Definition 5, is highly recursive. Notably, Rule (c) can feed into Rule (b), which can feed back into Rule (c). As a result, we have not implemented CP using techniques such as vector clocks, nor have we yet discovered a full CP implementation that only does online reasoning (*i.e.* never needs to "look back" in the execution trace); these remain challenging questions for future work. However, CP has an easy polynomial algorithm, derived directly from the definition and straightforwardly expressible in the Datalog language. Please see the POPL 2012 paper [139] for more information on the Datalog-based implementation.

The results presented here are on a collection of multithreaded Java benchmarks, mostly from previous studies [36, 51, 161]. The most substantial of our benchmarks are:

- Jigsaw, W3C's web server [154], coupled with a stress test harness.
- FtpServer, a high-performance FTP server implementation from The Apache Foundation [150], coupled with a JMeter workload [151].
- StaticBucketMap, a part of the Apache Commons project, offering a thread-safe

implementation of the Java Map interface. The code size of this benchmark is small, but its driver exercises it thoroughly, resulting in a long trace.

To perform the CP analysis on the benchmarks, we used the RoadRunner framework (Chapter 9) to dynamically instrument the bytecode of each benchmark at load time. The instrumentation code creates a stream of events for field and array accesses, synchronization operations, thread fork/joins, etc. We used this infrastructure to perform an inexpensive happens-before race analysis and to also produce a trace subsequently used for the post-mortem CP analysis. The CP analysis was thus explicitly coded to report races if they were not also HB races (since the latter were discovered and reported already). We conservatively translated accesses to volatile variables and thread creation/join events into pseudo-lock accesses. The traces produced were quite sizable even though stack-variable references are filtered out. The traces were then reduced to only maintain events concerning shared memory locations, and to eliminate re-accesses to the same variable by the same thread without intervening synchronization. Each reduced trace was then imported into a database and analyzed using a Datalog implementation [139]. Examining the intermediate results of our analysis indicates that CP is a much weaker relation than HB: the number of CP edges computed by our analysis (among synchronization operations) is typically as low as 10% to 20% of the number of HB edges for these benchmarks. Thus, happens-before race detection often considers events to be ordered when there is no semantic reason why they should be.

The first columns of Table A.1 show the main metrics for the benchmarks. The benchmark size in LoC is not entirely representative of its complexity: much of the code in a program's directory is library code, not exercised at all. Conversely, much of the code actually exercised is Java library code, never shown in the benchmark size.[1] StaticBucketMap is the most extreme example: if we were to report its code size uni-

[1]Library code is still valuable to analyze: the code may not contain races, but may be used in an unsafe way, exposing a race in client code.

Programs	Size (LoC)	Trace Size (#events)		Thread Count	Race Conditions Detected		
		Original	Reduced		HB (1 run)	HB (10 runs)	CP
banking	145	762	522	10	1	1	+0
elevator	1.4k	25k	16k	5	0	0	+0
FtpServer	39k	992k	543k	11	21	27	+7
hedc	25k	102k	1.4k	6	5	5	+0
Jigsaw	49k	1,992k	42k	77	18	33	+3
philo	86	669	382	6	0	0	+0
pool1.2	8.4k	692	526	8	0	0	+0
pool1.3	24k	841	683	8	0	0	+0
StaticBucketMap	see text	265k	133k	5	7	7	+0
stringBuffer	1.4k	223	178	8	0	0	+0
tsp	706	328k	381	4	0	0	+0
vector	26k	325	270	15	1	1	+0

Table A.1: Benchmark metrics and number of races detected by our CP analysis but not found in 10 HB runs.

formly with the other benchmarks, it would come to 110KLoC. The directory contains the entire Apache Commons Collections project, however. The main StaticBucketMap class and test driver file are just 807 LoC. Neither of the two sizes is representative of the code actually exercised, though the second is closer. Similar caveats apply to the report of thread counts. This metric lists the total number of thread created, which can be higher than the number of threads active simultaneously.

Table A.1 collects the experimental results. This table reports the races found in a single HB run, in 10 HB runs, as well as the races found by CP in its single run but never found in the 10 HB detector runs. Races are reported per-variable (*i.e.* dynamic race instances are collapsed based on which data words they occur on). Still, multiple races may have the same underlying cause (*e.g.* a single missing lock/unlock may fix more than one race).

Appendix B

CP Proof

In this appendix, we reproduce the soundness (precision) proof for the CP relation for reference. Please see the POPL 2012 paper for more information [139].

The statement of the theorem applies only to one CP race. (And, since $<_{CP}$ is a subset of $<_{HB}$, every HB-race is also a CP-race.) That is, the theorem proof establishes that either the "first" CP-race of a trace is an HB-race in some correct reordering of the trace or we can produce a correct reordering with a deadlock. (The idea of stating the soundness guarantee so that it applies to the first error reported is standard [62, 66].) The first race is the one that *finishes* earliest in the total order of the trace, *i.e.* a CP-race between events e_1 by t_1 and e_2 by t_2, with $e_1 <_{\text{TO}} e_2$, such that that there is no CP-race between two events both of which appear before e_2, as well as no race between events e_3-e_2, with e_3 appearing after e_1 and before e_2.

Although the theorem's guarantee applies to only one race, we can conservatively maintain soundness when reporting multiple races, at the cost of potentially missing some. Specifically, once a CP-race (which may be merely an HB-race) is discovered (and reported), the rest of the trace can be treated as if the CP-race were a CP edge, thus hard-ordering the two racy events. This means that the soundness guarantee of the theorem then applies to the *next* CP-race reported: any correct reordering of a

restricted trace (*i.e.* one with extra CP edges) is a correct reordering of the original trace. The drawback is that some CP-race nearby another CP race may not be reported due to our conservative treatment.

The soundness theorem proved below is subtle: if our detector issues a warning, there is either a correct reordering of the observed execution that exhibits an HB race, or a reordering that exhibits a deadlock. Thus, the race soundness guarantee only applies to deadlock-free programs, yet in practice a sound warning of a possible deadlock is just as valuable as a warning of a race.

Theorem (CP is Sound)**.** Given a trace tr with a CP-race, we can produce a $tr'' =_{CR} tr$ with either an HB-race or a deadlock.

Proof. Let the first race of trace tr be between events e_1 and e_2, with e_1 appearing before e_2 in the trace. Being the first race means that there is no CP-race between two events both of which appear before e_2, as well as no race between events e_3-e_2, with e_3 appearing after e_1 and before e_2.

Consider a trace tr' such that:

- $tr' =_{CR} tr$ and tr' has the same first CP-race as tr, *i.e.* between events e_1 by t_1 and e_2 by t_2, with $e_1 <_{\text{TO}}^{tr'} e_2$.
- Among traces satisfying the above property, tr' has minimal distance, in terms of the number of operations in $TO^{tr'}$ between events e_1 and e_2 of the first CP-race. (Intuitively, this means that all irrelevant events between e_1 and e_2 are correctly-reordered out of the e_1-e_2 segment in trace tr'.)
- Among traces that satisfy the above properties, tr' is one that also minimizes the distance between e_2 and every beginning of a critical section containing e_1, from innermost to outermost. That is, among traces that have the same (minimal) distance between e_2 and the innermost acquisition event k_1 for a critical section containing e_1,

tr' minimizes the distance between e_2 and the second such lock acquire, among those satisfying all the above tr' minimizes the distance between e_2 and the third such lock acquire, k_3, and so on, all the way to the outermost critical section containing e_1.

We will refer to the last two requirements as *the minimality property*. For such a trace tr', we get important lemmas:

Lemma 1. All events e between e_1 and e_2 are such that

(a) $e_1 <_{HB} e$ and $e <_{HB} e_2$

(b) $e_1 \not<_{CP} e$ and $e \not<_{CP} e_2$.

Proof. We prove each case separately.

(a) • Assume $e \not<_{HB} e_2$. Then we can move e and all events e' that occur between e and e_2 such that $e <_{HB} e'$ to the point right after e_2. The result of the move maintains program order. (Note that thread t_2 is not affected by the move at all, since we have already assumed that $e \not<_{HB} e_2$, therefore e cannot happen-before any previous event in t_2.) If this move (or any move in later proofs) is not possible it is because of one of two reasons:

- It causes the result to not be a well-formed trace because the pairing of lock acquisition/releases becomes invalid (*i.e.* a lock is acquired while held).
- The result is a valid trace t but $t \neq_{CR} tr'$ because a read now sees a different written value.

The former means that the moved events have a common lock with some non-moved event (say, f) that occurs before e_2—an impossibility since in this case $e <_{HB} f$, hence f would be a moved event. The latter reason is also an impossibility since then a moved event would clash with a non-moved event, f. If the two events were CP-ordered, then they would also be HB ordered, hence f

would have moved. If the events were not CP-ordered then e_1-e_2 would not have been the first race of tr'. Hence the move is possible, which violates the first minimality property, therefore our assumption was false and $e <_{HB} e_2$.

- If $e_1 \not<_{HB} e$, then we can move e and all events e' that occur between e_1 and e such that $e' <_{HB} e$ to right before e_1. Again, with similar reasoning as above (or as in Lemma 2, below) we get a contradiction.

(b) If either $e_1 <_{CP} e$ or $e <_{CP} e_2$, then we would have had $e_1 <_{CP} e_2$, per part (a) and the definition of CP: a contradiction.

□

Lemma 2. Consider any lock acquire event a_1 for a critical section containing e_1. All events e between a_1 (inclusive) and e_1 have $a_1 <_{HB} e$ and if $e_1 <_{HB} e_2$ then $e <_{HB} e_2$.

Proof. Assume $a_1 \not<_{HB} e$. Let E be the set of operations made up of e and all e' that occur between a_1 and e such that $e' <_{HB} e$. Note that set E cannot contain any events from thread t_1, or else $a_1 <_{HB} e$. Try to move all operations in E to right before a_1. If the move is not possible, it is either because these moved events have a common lock with some non-moved event, f (a contradiction, since then $f <_{HB} e$, and f would be moved) or that the moved events clash with a non-moved event (also a contradiction since it would imply a race before e_1-e_2 or a CP relation, which violates the assumption of no-HB between a_1 and the moved events). Therefore, moving E before a_1 is possible, and the result of the move maintains intra-thread order. However, moving E violates the minimality property of tr'.

The fact that ($e <_{HB} e_2$) follows from similar reasoning as in Lemma 1, but uses the assumption that $e_1 <_{HB} e_2$ to establish that e_1 is not among the moved events. □

Lemma 3. Any clashing events that both occur before e_2, or with one being e_2 and the other occurring after e_1 and before e_2, have to be CP-ordered.

Proof. Otherwise we trivially have a CP-race earlier than e_1-e_2. □

Lemma 4. Consider any lock acquire event a_1 for a critical section containing e_1. If a critical section $c...c'$ starts before event a_1 and ends after a_1 and before e_1 then $a_1 <_{CP} c'$.

Proof. By induction.

Base case: Consider the $c...c'$ that ends the soonest after a_1 (among all such $c...c'$ that satisfy the stated conditions). Assume that $a_1 \not<_{CP} c'$. By the well-nesting of lock operations, such a critical section $c...c'$ cannot be performed by thread t_1.

Let d be the first event after a_1 in this critical section. We will try to move $d...c'$ to the point right before a_1, respecting intra-thread order. If the move is successful it violates the minimality of tr', hence the move must be illegal because it violates some property of CR or of the definition of a trace. Therefore, the move must be illegal for either of the usual two reasons:

1. It causes the result to not be a trace because the pairing of lock acquisition/releases becomes invalid (*i.e.* a lock is acquired while held).
2. The result is a valid trace t but $t \neq_{CR} tr'$ because a read now sees a different written value.

For case (1), the moved events cannot be acquiring a lock held by thread t_1 at position a_1, since that lock would not be released before e_1. If the lock were held by a thread other than t_1, we have a critical section with the stated properties for $c...c'$ that ends before the currently considered $c...c'$, which is impossible.

Case (2) means that the moved events clash with some non-moved event that occurs after a_1. This non-moved event e'' has to CP the moved event it clashes with (by Lemma 3). But from Lemma 2 we have $a_1 <_{HB} e''$ and therefore $a_1 <_{CP} c'$.

Inductive case: the argument is identical to the base case, except in case (1) when we consider the possibility that a lock that needs to be acquired by the moved events is held by a thread other than t_1. In this case, we have an earlier critical section $g...g'$ with the stated properties, and therefore $a_1 <_{CP} g'$, by the inductive assumption. But since our $c...c'$ acquires the same lock, we get the desired $a_1 <_{CP} c'$.

□

Lemma 5. There cannot be a critical section by a thread other than t_2 that starts after event e_1 and before e_2, and ends after event e_2.

Proof. Assume that such critical sections exist. Among them pick the $c...c'$ that starts last, *i.e.* closest to e_2. Let d be the last event before e_2 of this critical section. We have two cases:

1. If $c...d$ does not contains nested critical sections inside it, we can move all events $c...d$ to the point right after e_2. The proof is similar to that of Lemma 4. The move respects intra-thread ordering. Also the moved events cannot be acquiring a lock held at point e_2. (There are no nested critical sections in the moved events, and the lock acquired by event c is still held at e_2.) Furthermore, the resulting trace is a correct reordering of the original because if it were not we would then have a clash between events whose relative position changes, *i.e.* between the moved events and non-moved events. But in that case there would be a CP edge originating in $c...d$ to an event before e_2 (by Lemma 3) and since c is between e_1 and e_2 we would get (using Lemma 1) $e_1 <_{CP} e_2$ (a contradiction).

2. If $c...d$ does contain critical sections, let $g...g'$ be the one ending last before e_2. Consider an event sequence produced as follows:

- we drop all events starting from (and including) g of that thread

- we drop all events after e_2 by all other threads.

Clearly the result is a prefix of tr'. If it is a legal trace that correctly reorders tr' then we are violating the minimality of tr'. In the resulting event sequence there cannot be an event acquiring a lock already held: the only dropped lock release events are either after e_2 (in which case subsequent lock acquisitions are also dropped), or are dropped together with their lock acquisition event (in the case of $g...g'$ events). Note that if there is a critical section inside $c...c'$ that starts before $g...g'$, it has to also end before g, or it would violate the definition of either $g...g'$ or $c...c'$. Also, no read can see a different write, or this would imply a clash between a dropped event after g and another before e_2. In such a case we would have a CP ordering, per Lemma 3 and $e_1 <_{CP} e_2$, as before. We conclude that the resulting trace correctly reorders tr' and violates its minimality assumption: a contradiction.

□

Armed with these lemmas about trace tr' we can now attempt to prove the soundness theorem. We will show that tr' either has an HB race (in fact, e_1 and e_2 have to be adjacent in this minimal trace) or, if not, the trace exposes a deadlock which can be caused by a slightly reordered trace.

Clearly, if e_1-e_2 is an HB race in tr' then we are done. Assume it is not. We will try to CR-reorder tr' so that one of the minimality properties is violated (which is a contradiction). Consider the first event f such that:

- f is performed by a thread other than t_1
- f occurs after e_1 and before e_2.

Such an event needs to exist if e_1-e_2 are HB-ordered. Furthermore, by Lemma 1, $e_1 <_{HB} f$, and since f is the first such event in any thread other than t_1, it needs to be a lock acquire. Consider then the critical section $f...f'$. There are two cases:

1. f' occurs before e_2. Let $f...f'$ be over lock l. We get two subcases:

   (a) l is not held by t_1 during e_1.

   Since $e_1 <_{HB} f$ and f is the first such event outside t_1, there must be a critical section over l after e_1 and before f. Let g be the lock acquire event of that critical section. g has to be an event by thread t_1, otherwise the definition of f would be violated (there would be another "first" event). Also, g has to be after e_1, by our assumption that l is not held during e_1. Consider a move of $f...f'$ to right before point g. The move respects intra-thread order. Also, if a read sees a different write then a moved event must clash with one of the non-moved events after g, hence (Lemma 3) we have some e'' such that $e'' <_{CP} f'$. But we have $e_1 <_{HB} e''$ (by Lemma 1), $e'' <_{CP} f'$, and $f' <_{HB} e_2$ (by Lemma 1), hence $e_1 <_{CP} e_2$: a contradiction.

   Finally, the move may cause a lock, m, to be acquired while being held: this means a critical section acquiring and releasing that lock is inside $f...f'$. (Assume w.l.o.g. that m is the first such lock.) If m is held at point g by a thread t_3, other than t_1, then it has to be released before f, violating the definition of f (since there is a different first event after e_1 by a thread other than t_1).

   A more interesting case is when lock m is held at point g by thread t_1. In that case, lock l is nested inside lock m in thread t_1 (because l is acquired at position g with m held) and lock m is nested inside lock l in thread t_2. We can cause a deadlock by moving a prefix of the $f...f'$ critical section (up until the lock m acquisition) to point g. Therefore the move of $f...f'$ to point

g either produces a legal trace t such that $t =_{CR} tr'$, or exposes a deadlock. The move can be repeated until there are no more critical sections over lock l between e_1 and $f...f'$. At that point, we can just move event f to right before e_1. This would produce a correctly reordered trace that violates the minimality of tr': a contradiction.

We conclude that if Case 1(a) occurs, there is always a deadlock in a correct reordering of trace tr'.

(b) l is held by t_1 during e_1.

Let a_2 be the last lock acquisition event of lock l before e_1. Consider a move of $f...f'$ and all previous events by the same thread after a_2 to right before point a_2. Let a_2' be the lock release paired with a_2. The move respects intra-thread order. Also, if a read sees a different write then a moved event must clash with one of the non-moved events after a_2, hence (by Lemma 3) we have some e'' such that $e'' <_{CP} e'$, where e' is a moved event, and therefore $e'' <_{CP} f'$ (since all the moved events precede f' and are by the same thread). But (by Lemma 2) we have $a_2 <_{HB} e''$ and therefore $a_2 <_{CP} f'$. Since, however, the critical sections starting at a_2 and ending at f' are over the same lock l, we get that $a_2' <_{CP} f'$, because of the second rule in the CP definition. This implies $e_1 <_{CP} e_2$ (since e_1 is before a_2', by assumption of case 1(b), and $f' <_{HB} e_2$, by Lemma 1): a contradiction.

Finally, we consider the case of the move being illegal because it causes a lock, m, to be acquired while being held. If such an m is held by a thread t_3, other than t_1, at point a_2, then it has to be released before e_1 (otherwise the release event would violate the definition of f, since it would come before it, after e_1 and by a thread other than t_1). This means that Lemma 4 applies to the critical section of that thread. Hence, we have that $a_2 <_{CP} h'$, where h' is

m's release event in t_3. But since h' happens-before some moved event (since the moved events acquire lock m), we get $e_1 <_{CP} f'$ (again, all moved events are program-ordered with f') and consequently (via Lemma 1) $e_1 <_{CP} e_2$: a contradiction.

If lock m is held at position a_2 by thread t_1, then l is nested inside m in thread t_1, while m is nested inside l in the thread performing $f...f'$. (The lock acquisition of m by that thread cannot be before f since the lock is released after e_1 and f is the first event after e_1 by a thread other than t_1. Therefore m is acquired and released inside critical section $f...f'$.) As before, we can cause a deadlock by moving a prefix of the $f...f'$ critical section (and any earlier events after a_2 by the same thread) to a_2.

Therefore, this case again implies a deadlock in a reordering of trace tr'.

2. f' occurs after e_2.

We then have by Lemma 5 that $f...f'$ has to be performed by thread t_2. Let $f...f'$ be over lock l. Lock l cannot be held by t_1 during event e_1, or $e_1 <_{CP} e_2$. Therefore, there must be some critical section $g...g'$ over l, performed by thread t_1 after e_1, such that $g' <_{HB} f$. (Recall that f is the first event after e_1 by a thread other than t_1.) Assume w.l.o.g. that $g...g'$ is the last such critical section.

Consider an event sequence produced as follows:

- we drop g and all events e' after g by a thread other than t_2.
- we drop all events after e_2 by all threads.

Clearly the result is a prefix of tr'. If it is a legal trace that correctly reorders tr' then we are violating the minimality of tr'. Therefore the result of this event drop has to be illegal. The drop respects intra-thread order. Also, if a read sees a different write then a dropped event must clash with one of the non-dropped

events before e_2. By Lemma 3 we get a CP edge between events after e_1 and before e_2 in tr', and by Lemma 1 and CP properties we have $e_1 <_{CP} e_2$: a contradiction.

Thus, the event sequence cannot be a trace: a lock has to be acquired while held. Such a lock, m, has to be acquired before one of the dropped events, with its release among the dropped events. The lock is then re-acquired by one of the non-dropped events, *i.e.* by thread t_2. The acquisition of m has to be in thread t_1 (otherwise f would not be the first event between e_1 and e_2 by a thread other than t_1). In thread t_1, for trace tr', lock l has to be nested inside m (since dropping every event after g, which is an acquisition of l, caused the drop of the release but not the acquisition of m). However, lock m is nested inside l in thread t_2, since it is acquired after l's acquisition (point f) and before l's release (which occurs after e_2). We can again cause a deadlock with an event move (of a prefix of $f...e_2$).

This concludes the proof of the theorem: any CP race implies either an HB race in the minimal trace tr', or a deadlock in a reordered trace. □

Part X

References

Bibliography

[1] Martín Abadi. Automatic mutual exclusion and atomicity checks. *Concurrency, Graphs and Models*, 5065:510–526, 2008.

[2] Martín Abadi, Cormac Flanagan, and Stephen N. Freund. Types for safe locking: Static race detection for Java. *Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):207–255, 2006.

[3] Martin Abadi and Gordon Plotkin. A model of cooperative threads. In *Symposium on Principles of Programming Languages (POPL)*, 2009.

[4] Sarita Adve. Data races are evil with no exceptions: Technical perspective. *Communications of the ACM*, 53:84–84, November 2010.

[5] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[6] Sarita V. Adve, Mark D. Hill, Barton P. Miller, and Robert H. B. Netzer. Detecting data races on weak memory systems. In *International Symposium on Computer Architecture (ISCA)*, 1991.

[7] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack management. In *USENIX Annual Technical Conference*, 2002.

[8] Rahul Agarwal, Amit Sasturkar, Liqiang Wang, and Scott D. Stoller. Optimized runtime race detection and atomicity checking using partial discovered types. In *International Conference on Automated Software Engineering (ASE)*, 2005.

[9] Alexander Aiken and David Gay. Barrier inference. In *Symposium on Principles of Programming Languages (POPL)*, 1998.

[10] Roberto M. Amadio and Silvano Dal Zilio. Resource control for synchronous cooperative threads. In *International Conference on Concurrency Theory (CONCUR)*, 2004.

[11] Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. In *International Workshop on Verification and Validation of Enterprise Information Systems*, 2003.

[12] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009.

[13] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In *Operating Systems Design and Implementation (OSDI)*, 2010.

[14] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: a dialect of Java without data races. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2000.

[15] Utpal Banerjee, Brian Bliss, Zhiqiang Ma, and Paul Petersen. A theory of data race detection. In *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD)*, 2006.

[16] S. Bensalem and K. Havelund. Dynamic deadlock analysis of multi-threaded programs. In *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD)*, 2005.

[17] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[18] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for Deterministic Parallel Java. Technical Report UIUCDCS-R-2009-3032, Department of Computer Science, University of Illinois at Urbana-Champaign, 2009.

[19] Eric Bodden and Klaus Havelund. Racer: effective race detection using AspectJ. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2008.

[20] Hans-J. Boehm. How to miscompile programs with "benign" data races. In *USENIX conference on Hot topics in Parallelism (HotPar)*, 2011.

[21] H.J. Boehm. Simple thread semantics require race detection. In *Fun and Interesting Thoughts (FIT) at PLDI*, 2009.

[22] S. Boroday, A. Petrenko, J. Singh, and H. Hallal. Dynamic analysis of Java applications for multithreaded antipatterns. In *International Workshop on Dynamic Analysis (WODA)*, 2005.

[23] Gérard Boudol. Fair cooperative multithreading. In *International Conference on Concurrency Theory (CONCUR)*, 2007.

[24] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. A type system for preventing data races and deadlocks in Java programs. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2002.

[25] Rhodes Brown, Karel Driesen, David Eng, Laurie Hendren, John Jorgensen, Clark Verbrugge, and Qin Wang. STEP: A framework for the efficient encoding of general trace data. In *Workshop on Program Anaylsis for Software Tools and Engineering (PASTE)*, 2002.

[26] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[27] J. Burnim and K. Sen. DETERMIN: Inferring Likely Deterministic Specifications of Multithreaded Programs. In *International Conference on Software Engineering (ICSE)*, 2010.

[28] Jacob Burnim and Koushik Sen. Asserting and checking determinism for multithreaded programs. In *International Symposium on Foundations of Software Engineering (FSE)*, 2009.

[29] RH Carver and K.C. Tai. Replay and testing for concurrent programs. *IEEE Software*, 8(2):66–74, 1991.

[30] CERN. Colt 1.2.0. `http://dsd.lbl.gov/~hoschek/colt/`, 2007.

[31] Fangzhe Chang and Jennifer Ren. Validating system properties exhibited in execution traces. In *International Conference on Automated Software Engineering (ASE)*, 2007.

[32] Feng Chen and Grigore Roşu. Predicting concurrency errors at runtime using sliced causality. Technical Report UIUCDCS-R-2006-2965, Department of Computer Science, University of Illinois at Urbana-Champaign, 2006.

[33] Feng Chen and Grigore Roşu. Mop: an efficient and generic runtime verification framework. *SIGPLAN Notices*, 42(10):569–588, 2007.

[34] Feng Chen and Grigore Roşu. Parametric and sliced causality. In *Computer Aided Verification (CAV)*, 2007.

[35] Feng Chen, Traian Florin Şerbănuţă, and Grigore Roşu. Effective predictive runtime analysis using sliced causality and atomicity. Technical Report UIUCDCS-R-2007-2905, Department of Computer Science, University of Illinois at Urbana-Champaign, October 2007.

[36] Feng Chen, Traian Florin Şerbănuţă, and Grigore Roşu. jPredictor: a predictive runtime analysis tool for Java. In *International Conference on Software Engineering (ICSE)*, 2008.

[37] J.D. Choi and H. Srinivasan. Deterministic replay of Java multithreaded applications. In *Symposium on Parallel and Distributed Tools (SPDT)*, 1998.

[38] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridhara. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Conference on Programming Language Design and Implementation (PLDI)*, 2002.

[39] Jong-Deok Choi, Barton P. Miller, and Robert H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):491–530, 1991.

[40] S.E. Choi and E.C. Lewis. A study of common pitfalls in simple multi-threaded programs. *ACM SIGCSE Bulletin*, 32(1):329, 2000.

[41] Mark Christiaens and Koenraad De Bosschere. TRaDe: Data Race Detection for Java. In *International Conference on Computational Science (ICCS)*, 2001.

[42] K.E. Coons, S. Burckhardt, and M. Musuvathi. GAMBIT: effective unit testing for concurrency libraries. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2010.

[43] Coverity. `http://www.coverity.com/`.

[44] Marcelo d'Amorim and Klaus Havelund. Event-based runtime verification of Java programs. In *International Workshop on Dynamic Analysis (WODA)*, 2005.

[45] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: deterministic shared memory multiprocessing. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[46] Anne Dinning and Edith Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 1990.

[47] Anne Dinning and Edith Schonberg. Detecting access anomalies in programs with critical sections. *SIGPLAN Notices*, 26(12):85–96, 1991.

[48] Matthew B. Dwyer and Lori A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *International Symposium on Foundations of Software Engineering (FSE)*, 1994.

[49] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded Java programs. *Concurrency and Computation: Practice and Experience*, 15:485–499, 2003.

[50] L. Eeckhout, K. de Bosschere, and H. Neefs. Performance analysis through synthetic trace generation. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2000.

[51] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: a race and transaction-aware Java runtime. In *Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[52] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamarić. Delay-bounded scheduling. In *Symposium on Principles of Programming Languages (POPL)*, 2011.

[53] Dawson R. Engler and Ken Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[54] Y. Eytani, R. Tzoref, and S. Ur. Experience with a Concurrency Bugs Benchmark. In *IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW)*, 2008.

[55] Y. Eytani and S. Ur. Compiling a benchmark of documented multi-threaded bugs. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[56] Azadeh Farzan and P. Madhusudan. Causal atomicity. In *Computer Aided Verification (CAV)*, 2006.

[57] Azadeh Farzan and P. Madhusudan. Monitoring atomicity in concurrent programs. In *Computer Aided Verification (CAV)*, 2008.

[58] Azadeh Farzan and P. Madhusudan. The complexity of predicting atomicity violations. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2009.

[59] Azadeh Farzan, P. Madhusudan, and Francesco Sorrentino. Meta-analysis for atomicity violations under nested locking. In *Computer Aided Verification (CAV)*, 2009.

[60] FindBugs. `http://findbugs.sourceforge.net/`.

[61] Cormac Flanagan and Stephen N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Symposium on Principles of Programming Languages (POPL)*, 2004.

[62] Cormac Flanagan and Stephen N. Freund. FastTrack: Efficient and precise dynamic race detection. In *Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[63] Cormac Flanagan and Stephen N. Freund. The roadrunner dynamic analysis framework for concurrent programs. In *Program Analysis for Software Tools and Engineering (PASTE)*, 2010.

[64] Cormac Flanagan, Stephen N. Freund, Marina Lifshin, and Shaz Qadeer. Types for atomicity: Static checking and inference for Java. *Transactions on Programming Languages and Systems (TOPLAS)*, 30(4):1–53, 2008.

[65] Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. Exploiting purity for atomicity. *IEEE Transactions on Software Engineering*, 31(4):275–291, 2005.

[66] Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[67] Scott D. Fleming. *Successful Strategies for Debugging Concurrent Software: An Empirical Investigation.* PhD thesis, Michigan State University, 2009.

[68] Emmanuel Fleury and Grégoire Sutre. Raja, version 0.4.0-pre4. Available at `http://raja.sourceforge.net/`, 2007.

[69] Dan Grossman. Type-safe multithreading in Cyclone. In *Workshop on Types in Language Design and Implementation (TLDI)*, 2003.

[70] R. Gustavsson. Ensuring dependability in service oriented computing. In *International Conference on Security & Management (SAM)*, 2006.

[71] Christian Hammer, Julian Dolby, Mandana Vaziri, and Frank Tip. Dynamic detection of atomic-set-serializability violations. In *International Conference on Software Engineering (ICSE)*, 2008.

[72] A. Hamou-Lhadj and T. C. Lethbridge. An efficient algorithm for detecting patterns in traces of procedure calls. In *International Workshop on Dynamic Analysis (WODA)*, 2003.

[73] Derin Harmanci, Pascal Felber, Vincent Gramoli, and Christof Fetzer. TMunit: Testing Transactional Memories. In *Workshop on Transactional Computing (TRANSACT)*, 2009.

[74] Jerry J. Harrow. Runtime checking of multithreaded applications with visual threads. In *International SPIN Workshop on Model Checking of Software*, 2000.

[75] A. Hartman, A. Kirshin, and K. Nagin. A test execution environment running abstract tests for distributed software. In *Conference on Software Engineering and Applications (SEA)*, 2002.

[76] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *Conference on History of Programming Languages (HOPL)*, 2007.

[77] Michael Isard and Andrew Birrell. Automatic mutual exclusion. In *Workshop on Hot Topics in Operating Systems (HOTOS)*, 2007.

[78] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Rosu, and D. Marinov. Improved multithreaded unit testing. In *International Symposium on Foundations of Software Engineering (FSE)*, 2011.

[79] V. Jagannath, M. Gligoric, D. Jin, G. Rosu, and D. Marinov. IMUnit: improved multithreaded unit testing. In *Workshop on Multicore Software Engineering*, 2010.

[80] Erwan Jahier, Mireille Ducassé, and Olivier Ridoux. Specifying prolog trace models with a continuation semantics. In *International Workshop on Logic Based Program Synthesis and Transformation (LOPSTR)*, 2001.

[81] Java Grande Forum. Java Grande benchmark suite. Available at `http://www.javagrande.org`, 2008.

[82] JLint. `http://artho.com/jlint/`.

[83] P. Joshi, C.S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[84] Shrinivas Joshi and Alessandro Orso. SCARPE: A technique and tool for selective capture and replay of program executions. In *International Conference on Software Maintenance (ICSM)*, 2007.

[85] Junit. `http://junit.org`.

[86] Vineet Kahlon, Franjo Ivančić, and Aarti Gupta. Reasoning about threads communicating via locks. In *Computer Aided Verification (CAV)*, 2005.

[87] Vineet Kahlon and Chao Wang. Universal causality graphs: A precise happens-before model for detecting bugs in concurrent programs. In *Computer Aided Verification (CAV)*, 2010.

[88] Baris Kasikci, Cristian Zamfir, and George Candea. Data races vs. data race bugs: telling the difference with portend. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[89] Klocwork. `http://www.klocwork.com/`.

[90] Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamarić. Static and precise detection of concurrency errors in systems code using smt solvers. In *Computer Aided Verification (CAV)*, 2009.

[91] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[92] James R. Larus and Ravi Rajwar. *Transactional Memory*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2006.

[93] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[94] NG Leveson and CS Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.

[95] John Levine, Tony Mason, and Doug Brown. *lex & yacc*. O'Reilly, 2nd edition, 1992.

[96] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.

[97] Brad Long, Daniel Hoffman, and Paul Strooper. Tool support for testing concurrent java components. *IEEE Transactions on Software Engineering*, 29(6):555–566, 2003.

[98] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *SIGPLAN Notices*, 43(3):329–339, 2008.

[99] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *Symposium on Principles of Programming Languages (POPL)*, 2005.

[100] Simon Marlow. *Happy, a parser-generator for Haskell.* http://www.haskell.org/happy.

[101] Simon Marlow. *A lexical analyser generator for Haskell.* http://www.haskell.org/alex.

[102] J.R. Matijevic and E.A. Dewell. Anomaly recovery and the mars exploration rovers. Technical report, Jet Propulsion Laboratory, National Aeronautics and Space Administration, 2006.

[103] Friedemann Mattern. Virtual time and global states of distributed systems. In *Workshop on Parallel and Distributed Algorithms*, 1989.

[104] John M. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Supercomputing*, 1991.

[105] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. *SIGPLAN Notices*, 42(6):446–455, 2007.

[106] Madanlal Musuvathi and Shaz Qadeer. Fair stateless model checking. In *Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[107] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Operating Systems Design and Implementation (OSDI)*, 2008.

[108] M. Naik, C.S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *International Conference on Software Engineering (ICSE)*, 2009.

[109] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[110] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM SIGPLAN Notices*, 42(6):100, 2007.

[111] Robert H. B. Netzer and Barton P. Miller. What are race conditions? Some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1:74–88, 1992.

[112] Hiroyasu Nishiyama. Detecting data races using dynamic escape analysis based on read barrier. In *Virtual Machine Research and Technology Symposium (VM)*, 2004.

[113] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2003.

[114] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[115] Chang-Seo Park and Koushik Sen. Randomized active atomicity violation detection in concurrent programs. In *International Symposium on Foundations of Software Engineering (FSE)*, 2008.

[116] Simon Peyton Jones. *A pretty printer library in Haskell.* Part of the GHC distribution at http://www.haskell.org/ghc.

[117] PMD. `http://pmd.sourceforge.net/`.

[118] Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multihreaded C++ programs. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2003.

[119] Eli Pozniansky and Assaf Schuster. MultiRace: Efficient on-the-fly data race detection in multithreaded C++ programs. *Concurrency and Computation: Practice and Experience*, 19(3):327–340, 2007.

[120] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Context-sensitive correlation analysis for detecting races. In *Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[121] William Pugh and Nathaniel Ayewah. Unit testing concurrent software. In *International Conference on Automated Software Engineering (ASE)*, 2007.

[122] Y. Qi, Y. Nir-Buchbinder, E. Farchi, R. Das, Z.D. Luo, and Z. Gan. Unit testing for concurrent business code. In *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD)*, 2010.

[123] M. Ricken and R. Cartwright. ConcJUnit: unit testing for concurrent programs. In *Conference on Principles and Practice of Programming in Java (PPPJ)*, 2009.

[124] Michiel Ronsse and Koenraad De Bosschere. RecPlay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems (TOCS)*, 17(2):133–152, 1999.

[125] Caitlin Sadowski, Thomas Ball, Judith Bishop, Sebastian Burckhardt, Ganesh Gopalakrishnan, Joseph Mayo, Shaz Qadeer, Madanlal Musuvathi, and Stephen Toub. Practical parallel and concurrent programming. In *Symposium on Computer Science Education (SIGCSE)*, 2011.

[126] Caitlin Sadowski, Stephen N. Freund, and Cormac Flanagan. SingleTrack: A dynamic determinism checker for multithreaded programs. In *European Symposium on Programming (ESOP)*, 2009.

[127] Caitlin Sadowski and Jaeheon Yi. Tiddle: A trace description language for generating concurrent benchmarks to test dynamic analyses. In *International Workshop on Dynamic Analysis (WODA)*, 2009.

[128] Caitlin Sadowski and Jaeheon Yi. Applying usability studies to correctness conditions: A case study of cooperability. In *Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, 2010.

[129] M. Said, C. Wang, Z. Yang, and K. Sakallah. Generating data race witnesses by an SMT-based analysis. In *NASA Formal Methods Symposium*, 2011.

[130] Amit Sasturkar, Rahul Agarwal, Liqiang Wang, and Scott D. Stoller. Automated type-based analysis of data races and atomicity. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2005.

[131] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4), 1997.

[132] Edith Schonberg. On-the-fly detection of access anomalies. In *Conference on Programming Language Design and Implementation (PLDI)*, 1989.

[133] Viktor Schuppan, Marcel Baur, and Armin Biere. JVM independent replay in Java. *Electronic Notes in Theoretical Computer Science*, 113:85–104, 2005.

[134] Koushik Sen. Race directed random testing of concurrent programs. In *Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[135] Koushik Sen and Gul Agha. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In *IIFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, 2005.

[136] Koushik Sen, Grigore Rosu, and Gul Agha. Online efficient predictive safety analysis of multithreaded programs. *International Journal on Software Technology and Tools Transfer (STTT)*, 8(3):248–260, 2006.

[137] Traian Florin Şerbănuţă, Feng Chen, and Grigore Roşu. Maximal causal models for multithreaded systems. Technical Report UIUCDCS-R-2008-3017, University of Illinois at Urbana-Champaign, Department of Computer Science, December 2008.

[138] Nishant Sinha and Chao Wang. On interference abstractions. In *Symposium on Principles of Programming Languages (POPL)*, 2011.

[139] Yannis Smaragdakis, Jacob M. Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. Sound predictive race detection in polynomial time. In *Symposium on Principles of Programming Languages (POPL)*, 2012.

[140] Yannis Smaragdakis, Anthony Kay, Reimer Behrends, and Michal Young. Transactions with isolation and cooperation. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007.

[141] Yannis Smaragdakis, Anthony Kay, Reimer Behrends, and Michal Young. General and efficient locking without blocking. In *Workshop on Memory Systems Performance and Correctness (MSPC)*, 2008.

[142] Francesco Sorrentino, Azadeh Farzan, and P. Madhusudan. PENELOPE: weaving threads to expose atomicity violations. In *International Symposium on Foundations of Software Engineering (FSE)*, 2010.

[143] SPEC. Standard Performance Evaluation Corporation JBB2000 Benchmark. Available at `http://www.spec.org/osg/jbb2000/`, 2000.

[144] Standard Performance Evaluation Corporation. SPEC benchmarks. http://www.spec.org/, 2003.

[145] Nicholas Sterling. Warlock: A static data race analysis tool. In *USENIX Winter Technical Conference*, 1993.

[146] John Steven, Pravir Chandra, Bob Fleck, and Andy Podgurski. jrapture: A capture/replay tool for observation-based testing. *SIGSOFT Software Engineering Notes*, 25(5):158–167, 2000.

[147] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs Journal*, 30(3):16–20, 2005.

[148] G. Szeder. Unit testing for multi-threaded Java programs. In *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD)*, 2009.

[149] Sriraman Tallam, Chen Tian, and Rajiv Gupta. Dynamic slicing of multithreaded programs for race detection. In *IEEE International Conference on Software Maintenance (ICSM)*, 2008.

[150] The Apache Software Foundation. Apache FtpServer. Available at http://mina.apache.org/ftpserver/, 2009.

[151] The Apache Software Foundation. Apache JMeter. Available at http://jakarta.apache.org/jmeter/, 2009.

[152] The Eclipse Foundation. SWTBot. Available at http://www.eclipse.org/swtbot/, 2009.

[153] The Eclipse Foundation. The Eclipse Software Development Kit. Available at http://www.eclipse.org/, 2009.

[154] The World Wide Web Consortium. Jigsaw Web Server. Available from `http://www.w3.org/Jigsaw/`, 2009.

[155] Mohit Tiwari, Shashidhar Mysore, and Timothy Sherwood. Quantifying the potential of program analysis peripherals. In *Parallel Architectures and Compilation Techniques (PACT)*, 2009.

[156] Takayuki Usui, Yannis Smaragdakis, Reimer Behrends, and Jacob Evans. Adaptive locks: Combining transactions and locks for efficient concurrency. In *Parallel Architectures and Compilation Techniques (PACT)*, 2009.

[157] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *Symposium on Principles of Programming Languages (POPL)*, 2006.

[158] Willem Visser, Klaus Havelund, Guillaume P. Brat, and Seungjoon Park. Model checking programs. In *International Conference on Automated Software Engineering (ASE)*, 2000.

[159] R. Von Behren, J. Condit, F. Zhou, G.C. Necula, and E. Brewer. Capriccio: scalable threads for internet services. *ACM SIGOPS Operating Systems Review*, 37(5):281, 2003.

[160] Christoph von Praun and Thomas Gross. Object race detection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2001.

[161] Christoph von Praun and Thomas Gross. Static conflict analysis for multi-threaded object-oriented programs. In *Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[162] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. Relay: static race detection on

millions of lines of code. In *International Symposium on Foundations of Software Engineering (FSE)*, 2007.

[163] Chao Wang, Sudipta Kundu, Malay Ganai, and Aarti Gupta. Symbolic predictive analysis for concurrent programs. In *World Congress on Formal Methods (FM)*, 2009.

[164] Chao Wang, Rhishikesh Limaye, Malay Ganai, and Aarti Gupta. Trace-based symbolic analysis for atomicity violations. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2010.

[165] Liqiang Wang and Scott D. Stoller. Static analysis of atomicity for programs with non-blocking synchronization. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2005.

[166] Liqiang Wang and Scott D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2006.

[167] Liqiang Wang and Scott D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Transactions on Software Engineering*, 32:93–110, February 2006.

[168] Chunyang Ye, S. C. Cheung, W. K. Chan, and Chang Xu. Detection and resolution of atomicity violation in service composition. In *International Symposium on Foundations of Software Engineering (FSE)*, 2007.

[169] Jaeheon Yi, Tim Disney, Stephen N. Freund, and Cormac Flanagan. Types for precise thread interference. In *International Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2011.

[170] Jaeheon Yi and Cormac Flanagan. Effects for cooperable and serializable threads. In *Workshop on Types in Language Design and Implementation (TLDI)*, 2010.

[171] Jaeheon Yi, Caitlin Sadowski, and Cormac Flanagan. Sidetrack: Generalizing dynamic atomicity analysis. In *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD)*, 2009.

[172] Jaeheon Yi, Caitlin Sadowski, and Cormac Flanagan. Cooperative reasoning for preemptive execution. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2011.

[173] Yuan Yu, Tom Rodeheffer, and Wei Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2005.