

SHRINKWRAP: Efficient Dynamic Race Detection for Array-Intensive Programs

by

James Wilcox

Stephen N. Freund, Advisor

A thesis submitted in partial fulfillment
of the requirements for the
Degree of Bachelor of Arts with Honors
in Computer Science

Williams College
Williamstown, Massachusetts

May 16, 2013

Abstract

We explore a new technique for efficient dynamic race detection on programs using arrays intensively. Standard techniques lead to redundant operations and redundant representations in many common cases. For these common cases, we design dynamic compression methods that eliminate this redundancy. Finally, we implement our techniques in a prototype tool called SHRINKWRAP, which is built as an extension to a state-of-the-art precise dynamic race detector. We evaluate the performance and precision of SHRINKWRAP on a suite of benchmark programs.

We show that our prototype can improve performance dramatically when the target program accesses arrays in a pattern we recognize. The vast majority of the accesses that must be checked by the underlying race detector can be eliminated on almost half of our benchmark programs. However, we also find that our prototype is not always as time efficient as one might expect given the number of accesses eliminated.

Acknowledgments

I would like to thank Steve Freund, my advisor, for all of his guidance and patience, and for many enjoyable bike rides in good weather. Thanks also to Duane Bailey, my second reader, for his useful comments and enlightening perspective, and for making me a computer scientist two years ago. I am also extremely grateful to the computer science department as a whole for providing a challenging and exciting environment for me at Williams. Finally, I would like to thank my family and friends, both at Williams and elsewhere, for keeping me sane during my four years here.

Contents

1	Introduction	5
1.1	Contributions	6
1.2	Overview	7
2	Background	9
2.1	Race Conditions	9
2.2	Race Detection	16
2.3	Race Detection on Arrays	26
3	Summary of Analysis	33
3.1	Overview	33
3.2	Adaptive Array Representation	34
3.3	Dynamic Mode Inference and Race Checks	37
3.4	Expected Improvements	42
4	Analysis	45
4.1	Language	45
4.2	Unoptimized Detector Semantics	47
4.3	Optimized Detector Semantics	49
4.4	Equivalence of the Two Semantics	56
5	Implementation and Evaluation	63
5.1	Implementation	63
5.2	Evaluation	65
6	Conclusions	71
6.1	Contributions	71
6.2	Future Work	72
	Bibliography	73

Chapter 1

Introduction

The general-purpose computer market, from servers to tablets, is now dominated by systems that employ multicore processors, which have the ability to execute several instructions at once. This power sits idle unless operating systems and applications are written with concurrency in mind. Unfortunately, multithreaded and concurrent programming is inherently harder than its sequential counterpart because threads running concurrently may interact in unanticipated ways. In particular, concurrent programs can suffer from race conditions, which occur when two or more threads access and modify data at the same time without proper synchronization. These races lead to errors which depend on the relative timing of when threads execute the accesses. Since the threads do not properly synchronize, there is no guarantee that the same relative order of operations is used when the program is run multiple times. Thus, these bugs may manifest only occasionally, making them surprising, hard to diagnose, and hard to eliminate.

Because these bugs are so difficult to detect and eliminate, many techniques have been designed to detect them automatically. Static analyses examine the source code of the target program offline and attempt to prove that no races occur on any input to the program [1, 14]. These static techniques provide absolute guarantees of race freedom, but they are inherently conservative and thus produce many false positives on real-world programs. On the other hand, dynamic analyses monitor programs as they run and report any race conditions that occur [7, 11]. Dynamic techniques can be precise, reporting no false positives while reporting all real errors that occur on the observed execution. However, dynamic analyses only reason about a single input to the program, so they give no guarantees about the program when it is run on other

inputs. Although they provide weaker guarantees, dynamic analyses are the most promising direction forward, given the imprecision of even the best static analyses.

Dynamic race detectors also slow down the target program because they perform extra checking to detect race conditions. Detectors incur two types of overhead. First, the checks performed by the detector take time that could have otherwise been used by the target program. Second, for each memory location used by the program, the race detector must keep extra state. State-of-the-art techniques can lead to slow-downs of 5-10 times the unmonitored running time and memory overhead of 2-4 times the unmonitored space [7].

The standard technique for handling arrays in dynamic detectors is to treat each element independently of the others. Under this approach, programs using large arrays intensively are especially susceptible to massive overhead because the detector allocates state for each element of the array individually. This can easily cause a program’s memory requirements to exceed the maximum available memory on the system, meaning that race detection cannot be used. Even when analysis is possible, the additional state may still lead to degraded cache and memory performance, especially when the target program has been engineered to take advantage of caching.

1.1 Contributions

The goal of this thesis is to reduce the overhead of race detection on array-intensive programs. Although overhead is unavoidable in general, there is potential to mitigate it much of the time by exploiting a key insight: many threads access arrays in easy-to-describe patterns such as “touches every other element of the array.” By reasoning about these patterns, instead of considering each access individually, we can improve the efficiency of a detector. In order to be useful for race detection, these patterns must have the additional property that their elements are accessed without intervening synchronization. This ensures that no precision is lost if we treat all elements captured by a pattern as a single “abstract” element inside the detector and allocate only one state to represent the entire access pattern. It also allows us to perform only one check to determine whether any elements of a pattern were involved in a race.

The contributions of this thesis are as follows.

- We identify several common patterns in how programs access arrays and show how these patterns lead to redundancy in modern dynamic race detectors.

- We design an analysis that augments any precise dynamic race detector, eliminating the redundancy discussed above. We formalize the analysis and prove that it retains precision.
- We implement the analysis in the tool SHRINKWRAP.
- We evaluate SHRINKWRAP on a suite of benchmark programs and present performance data. We find that SHRINKWRAP is able to eliminate the vast majority of accesses that must be checked for race conditions.
- Finally, we provide some directions for future work, as well as reflecting on the overall insights gained throughout this work.

1.2 Overview

Chapter 2 introduces race conditions and reviews the literature on detecting race conditions. We use the happens-before relation to precisely define race conditions and as a conceptual framework for comparing different race detectors [11]. We also introduce vector clocks as an efficient way of performing happens-before queries [12].

Chapter 3 designs a simple description of access patterns that are common in practice. We discuss the properties required of these patterns in order to be useful for optimizing race detection. We also show how to use the patterns to compress state and eliminate redundant checks.

Chapter 4 gives the details of our analysis. We show how to optimize a given race detection algorithm for array-intensive programs using our patterned access analysis, including both compression and redundancy elimination. We also prove that the optimization is correct with respect to the underlying algorithm, in the sense that our analysis misses no new races and introduces no new false positives.

Chapter 5 presents and evaluates our implementation, SHRINKWRAP, which is an optimized version of FASTTRACK that uses the ideas from Chapters 3 and 4. We also validate the implementation on a set of benchmark programs and present the results. We point out programs with access patterns that can be well-described by our technique. We also consider programs that have patterns

to their accesses that we cannot capture, as well as programs whose accesses are random.

Chapter 6 draws conclusions and proposes several ideas for future work.

Chapter 2

Background

In this chapter, we study race conditions and their detection by program analysis. Section 2.1 introduces and defines race conditions over traces in terms of the happens-before relation. Section 2.2 reviews the large body of work that exists on detecting race conditions. Finally, Section 2.3 discusses previous techniques for detecting races in programs that use arrays.

2.1 Race Conditions

Multithreaded programs are prone to a wider class of bugs than their sequential counterparts. In particular, multithreaded programs suffer from race conditions. An execution of a multithreaded program can be thought of as an interleaving of operations from each thread. This interleaving can change from execution to execution due to the realities of modern multicore machines and their operating systems. Intuitively, a race occurs when two operations accessing the same variable could have been interleaved in a different order, where at least one operation is a write.

Race conditions are especially problematic because they can be difficult to reproduce, detect, and eliminate. Some races may occur only on rare interleavings, thus a program with races may behave correctly much of the time, for example during testing, but then fail when the rare case happens after deployment. A race-free program uses synchronization to force the desired order of operations between threads.¹ Even

¹It is important to note that a program without race conditions is not necessarily correct, due to the presence of other errors, but we focus on the elimination of races in this thesis. See Example 2.5 for a race-free program that still has concurrency errors.

	Thread 1	Thread 2
1	$t1 = x$	$t2 = x$
2	$t1 = t1 + 1$	$t2 = t2 + 1$
3	$x = t1$	$x = t2$

Figure 2.1: Incrementing program with race conditions. The shared variable x can be incremented once or twice depending on the interleaving of operations from each thread.

once the presence of a race condition has been detected, it is notoriously difficult to correct the error without introducing further concurrency bugs, because of the large number of possible interleavings that must be considered. These hazards make tools to detect race conditions extremely useful. For example, race detectors can be used to confirm that races have been eliminated. The development of such tools has been an active subject of research for some time, and we review the relevant work in this area in Section 2.2.

Before defining race conditions formally, we present a small example program that suffers from race conditions.

Example 2.1 (Simple Race Condition: Broken Increment). Consider the program in Figure 2.1.² Here, two threads attempt to increment the shared variable x without proper synchronization. If line 1 executes in both threads before line 3 executes in either thread, only one of the increments will be recorded on x . We will show below that our formal definition of a race condition identifies this (rather trivial) case. We'll also see a few ways to add synchronization to eliminate the race condition, as well as some more substantial examples.

Operations, Traces, and the Happens-Before Relation

In this section, we will refine the intuition given above into a precise definition of race conditions in terms of the happens-before relation on a trace. First, we introduce a simple set of operations sufficient to study race detection. Next, we introduce traces, which capture a particular interleaving of operations for a program. We then define a race on a trace as a pair of concurrent, conflicting accesses. Finally, we define

²In this example and for the rest of this chapter, we will use the notation of Figure 2.1 without definition. Programs written in this way are not part of the formalism studied below, since traces contain only the abstract operations listed in Table 2.1. Instead, these programs are meant to make clear the goal of each program, before diving into its traces.

Operation	Description
$\text{rd}(t, x)$	thread t reads from memory location x
$\text{wr}(t, x)$	thread t writes to memory location x
$\text{acq}(t, m)$	thread t acquires a lock m
$\text{rel}(t, m)$	thread t releases a lock m

Table 2.1: List of program operations relevant to race detection.

the happens-before relation over a trace, which formalizes the notion of concurrency. Throughout this section, we use the program from Example 2.1 to illustrate each concept. We also consider two ways of eliminating that program’s race conditions.

Operations. When studying data race detection techniques, many of the details of the computation performed by program can be ignored. Thus, in this discussion, we consider only the operations listed in Table 2.1. This list does not exhaustively represent the synchronization idioms used in practice; however, it is straightforward to extend our discussion to other synchronization operations, such as fork-join and volatile accesses.

Traces and Races. A particular execution of a program defines a *trace*, which records an interleaving of operations from each thread that is consistent with that execution. A trace contains a *race condition* if it contains a pair of concurrent, conflicting accesses. Accesses are *conflicting* if both accesses refer to the same memory location and at least one of them writes to that location. A pair of accesses is *concurrent* if, given sufficient resources, they could happen at the same time. More formally, accesses are concurrent if they are not ordered by the happens-before relation, which we define and discuss below.

Example 2.2 (Traces of Broken Increment). We have already mentioned that many computational aspects of target programs can be ignored. In Example 2.1, line 2 in each thread may be ignored if we do not consider the temporary variables to be shared memory locations. With this in mind, we can list the operations each thread of the program will perform.

Thread 1	Thread 2
$\text{rd}(1, x)$	$\text{rd}(2, x)$
$\text{wr}(1, x)$	$\text{wr}(2, x)$

```

Trace A:
  Thread 1  Thread 2
    rd(1,x)
    wr(1,x)
                rd(2,x)
                wr(2,x)

```

```

Trace B:
  Thread 1  Thread 2
    rd(1,x)
                rd(2,x)
    wr(1,x)
                wr(2,x)

```

```

Trace C:
  Thread 1  Thread 2
    rd(1,x)
                rd(2,x)
                wr(2,x)
    wr(1,x)

```

Figure 2.2: Three traces for the program from Example 2.1.

There can be many traces for a given program, and in general, there will be exponentially many in the length of the program. For this short program, it isn't too hard to enumerate all of them, but for brevity we give just a few representative examples in Figure 2.2.

Notice that the correct (twice-incremented) value of x is computed in trace A since thread 2's read operation occurs later than thread 1's write. However traces B and C were not so lucky; they both compute the incorrect (once-incremented) value. The fact that some traces compute the correct value can make discovering race conditions during testing difficult, especially since some traces may occur rarely, if at all. Adequate testing thus requires extremely careful exploration of the space of possible traces.

The Happens-Before Relation. Given a trace α , the *happens-before relation*, denoted \prec_α , is a partial order on operations in α . We will define \prec_α by specifying some pairs of operations that induce happens-before order in the trace and then requiring that the relation be transitively closed.

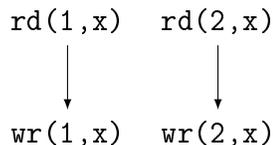


Figure 2.3: Happens-before relation for all three traces from Example 2.2 and Figure 2.2. Two operations are ordered by happens-before if and only if there is a directed path from one to the other in the figure. Since there are no synchronization operations in the program, the happen-before order is just the program order.

Program Order. Pairs of operations that occur in the same thread always happen in a trace in the order specified by the programmer, and we say that such operations are ordered in the happens-before relation by the *program order*.

Synchronization Order. Within any trace, a release operation on a lock followed by an acquire operation on the same lock by another thread can not appear in the opposite order, since a thread cannot acquire a lock that is already held. We say that these operations are ordered by the *synchronization order*.³

Finally, we define \prec_α to be the transitive closure of the union of the program order and the synchronization order. Note that this relation is not reflexive.

Example 2.3 (Happens-Before Relation for Broken Increment). Now let us consider the happens-before relation for each of the traces from Example 2.2. We can represent the happens-before relation graphically as in Figure 2.3.

In all three cases, the program order is the same: the write in each thread follows the read in that thread. Furthermore, there are no synchronization operations, so the synchronization order is empty. Thus the happens-before ordering of operations in this program is simply what is induced by the program ordering of each thread. In particular the happens-before relation is the same for all three traces. Since no operation in one thread is related by the program order to an operation in another thread, the same holds for the happens-before relation. Because the relative order of $\text{wr}(1, \mathbf{x})$ and any access to \mathbf{x} in thread 2 is not determined, there is the potential to compute inconsistent values. In other words, the program has race conditions. Note that conflicting accesses involve a write operation.

³If we had considered additional synchronization primitives, they would also contribute to the synchronization order. For example, a fork operation happens before the first operation of any thread it creates, and a join happens after the last operation of any thread it waits for.

	Thread 1:	Thread 2:
1	acquire m	acquire m
2	t1 = x	t2 = x
3	t1 = t1 + 1	t2 = t2 + 1
4	x = t1	x = t2
5	release m	release m

Figure 2.4: Race-free version of program from Figure 2.1.

The happens-before relation exposes a race in trace A, *even though the correct value was computed*. Indeed, it turns out that every possible trace of this program has a race that will be exposed by the happens-before relation. This robustness to changes in the trace is an important advantage of happens-before analysis over testing-based approaches.

Race-Free Programs. We will now give two ways to modify the program from Example 2.1 so that it is race free. We have seen that the program suffered from race conditions because there were no synchronization operations in the program, and thus no synchronization order contributions to the happens-before order. To remedy this, we must add synchronization and ensure the relative order of writes in one thread with accesses in another is consistent.

Example 2.4 (Correct Increment Program). Thinking about what goes wrong in traces that compute the once-incremented value, we realize that each thread assumes the value of x remains constant between the read and the write, but this assumption was not enforced by the (lack of) synchronization in the program. We can prevent multiple threads from accessing x at once by protecting each increment operation with a lock, say m . This leads to the program in Figure 2.4.

Since one thread will acquire the lock before the other and then run uninterrupted until the release, the two increments of x are serialized, as desired. The two possible traces of this program are recorded in Figure 2.5, and the happens-before relation for trace A is depicted in Figure 2.6.

Example 2.5 (Race-free, Incorrect Increment Program). We now consider another way of modifying the increment program to eliminate race conditions. In the first race-free version from Example 2.4, the locking has prevented the two threads from running in parallel. Wishing to restore as much of this parallelism as possible, we can bring the addition operations outside the locked sections, yielding the program

Trace A:		Trace B:	
Thread 1	Thread 2	Thread 1	Thread 2
acq(1,m)			acq(2,m)
rd(1,x)			rd(2,x)
wr(1,x)			wr(2,x)
rel(1,m)			rel(2,m)
	acq(2,m)	acq(1,m)	
	rd(2,x)	rd(1,x)	
	wr(2,x)	wr(1,x)	
	rel(2,m)	rel(1,m)	

Figure 2.5: The two possible traces for the program from Figure 2.4.

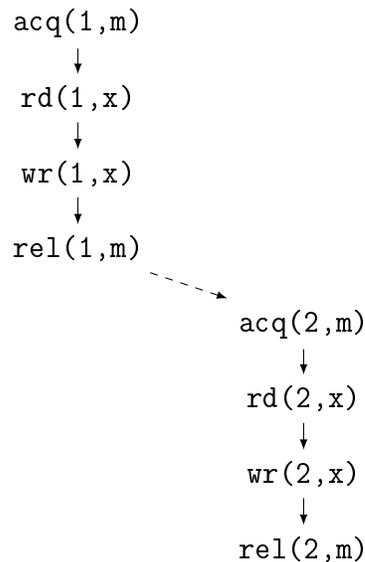


Figure 2.6: The happens-before relation for trace A from Figure 2.5. Edges from the program order are solid, while edges from the synchronization order are dashed. As before, operations are ordered by the happens-before relation if and only if there is a directed path from one to the other. In this case, all pairs of distinct operations are ordered. In particular, no access to x from one thread is concurrent with a write from another thread; there are no races.

in Figure 2.7. Consider any trace of this program. It is not difficult to see that all accesses to x are ordered by the happens-before relation: whichever thread acquires m first must complete its access while the other thread waits or increments.

We should now ask ourselves whether this program is correct? Certainly, there are no longer any race conditions, but consider the trace in Figure 2.8. For this trace,

	Thread 1:	Thread 2:
1	acquire m	acquire m
2	t1 = x	t2 = x
3	release m	release m
4	t1 = t1 + 1	t2 = t2 + 1
5	acquire m	acquire m
6	x = t1	x = t2
7	release m	release m

Figure 2.7: Race-free, incorrect increment program, whose synchronization prevents races but still allows bad interleavings, like the one in Figure 2.8.

Thread 1	Thread 2
acq(1,m)	
rd(1,x)	
rel(1,m)	
	acq(2,m)
	rd(2,x)
	rel(2,m)
acq(1,m)	
wr(1,x)	
rel(1,m)	
	acq(2,m)
	wr(2,x)
	rel(2,m)

Figure 2.8: Trace of program from Figure 2.7 that computes the wrong value.

only one increment will be applied because both reads happened before either write. The moral here is that lack of race conditions does not imply lack of concurrency bugs.

In the next section, we will review several algorithms for detecting race conditions in programs. The happens-before relation is a useful tool for understanding these analyses, even when the algorithms in question do not explicitly use it.

2.2 Race Detection

This section briefly reviews the literature on race detection. We first discuss program analysis in general, including goals and design tradeoffs. Then we review in some detail several state-of-the-art algorithms that have been proposed to detect race con-

ditions in programs. Finally, we introduce a simple dynamic analysis based on vector clocks, which will serve as the prototypical algorithm to be optimized in later chapters.

Program Analysis

A program analysis computes some property of its target program. Such analyses can be used to identify opportunities for optimization or to enforce correctness. Program analyses inhabit a large design space, where many techniques are used and tradeoffs are made, all of which impact the precision, performance, and usability of the analysis. We now review a few of the most important design choices with an eye towards analyses that enforce correctness.

Static vs. Dynamic. *Static analyses* process the source code of the target program and are inherently conservative because of the undecidability of the problems they solve. On the other hand, *dynamic analyses* observe the target program as it runs, and so they only reason about the execution of the program on a particular input, rather than in general. To be useful, dynamic analyses must be applied to enough different inputs so that many of the code paths are exercised. This is analogous to the problem of adequate coverage in a test suite.

Soundness and false negatives. An analysis produces a *false negative* if it marks an erroneous operation as correct. An analysis that is guaranteed to never produce false negatives is *sound*. For example, if an analysis accepted a program with races as race free, then it would not be sound. For static analyses, erroneous means that there is some input that would cause the error, while for dynamic analyses, erroneous simply means that the error occurred on this input.

Completeness and false positives. An analysis produces a *false positive* if it marks an operation as erroneous even though the operation was in fact correct. An analysis that is guaranteed to never produce false positives is *complete*. For example, if an analysis rejected a race-free program as having a race condition, then the analysis would not be complete. As above, a static analysis must analyze correctness with respect to all possible inputs, while dynamic analyses are restricted to the current input.

We say that a dynamic analysis is *precise* if it is sound and complete as described above. Note that precise static analysis is undecidable.

Race Detection Algorithms

At a high level, race detectors seek to find race conditions: pairs of concurrent, conflicting accesses. To do so, they must reason about the happens-before relation in some way. The precision and performance of the algorithm greatly depends on how it represents the happens-before relation. Maintaining enough information to determine whether two arbitrary operations in a trace are ordered by the happens-before relation is prohibitively expensive, so analyses typically approximate the happens-before relation in some way.

Static Analyses. Static analyses use inherently imprecise approximations to the happens-before relation because the problem is undecidable in general.

Chord seeks to soundly report all pairs of memory accesses that may be involved in a race [14]. To this end, it initially computes a simple over-approximation to the set of such pairs that rules out those pairs that access fields of different types. Chord then refines this approximation with further analysis, including an alias analysis and an analysis designed to take advantage of Java’s lexically scoped locking idiom. For soundness, Chord additionally employs a new form of alias analysis, conditional must-not-alias analysis, which seeks to prove that two locations are not aliased given that two other locations are not aliased. For example, the analysis can easily prove that the locations $x.f$ and $y.f$ are distinct given that locations x and y are distinct [13]. After further analysis, any remaining pairs are reported as potential races to the user.

RCC/JAVA is a type system for Java that is designed so that well-typed programs are race free. Like Chord, RCC/JAVA is sound. The type system tracks the set of locks known to be held at each program point and checks that the proper locks are held on each access. Early work required annotations specifying which fields required which locks [4], and later improvements developed algorithms to infer these annotations [5]. Program points that do not pass the required locking check cannot be typed, and the system reports an error. One advantage of a type system approach is that methods can be analyzed independently, supporting modularity. On the other hand, the system only supports lock-based synchronization and will report false positives on programs written with other idioms.

RacerX checks C programs for races using a static lockset analysis. In contrast to the two analyses above, RacerX is neither sound nor complete, instead focusing on scalability and usability issues. It approximates the set of locks held using a

context-sensitive, path-insensitive algorithm to add and remove locks when certain user-declared functions are called. The problem is made harder since C does not have lexically-scoped locking. RacerX also uses advanced filtering and sorting techniques, trading false negatives for fewer false positives. Its main advantage is scalability, as it is able to run on systems as large as the Linux kernel. Its filtering techniques also strive to make a system that is useful, rather than one of only theoretical interest. However, it is not able to provide the same guarantees as the systems above, since there may be further races even when none are reported.

Dynamic Analyses. We begin with some terminology. Dynamic analyses typically associate extra state to each memory location used by the program. We refer to this state as the *shadow state* for the program. When emphasizing the distinction between the analysis and the program being analyzed, we may refer to the program being analyzed as the *target program*.

We can categorize dynamic analyses by how they represent the happens-before relation. Broadly speaking, an analysis can use either a direct or indirect representation.

Direct Representation via Vector Clocks. Analyses may check the ordering of accesses by storing vector clocks [12] for each thread, lock, and memory location. This approach exploits the fact that once races have been ruled out on a particular operation, we no longer need to be able to answer happens-before queries for that operation. Keeping clocks relevant to all recent operations by threads retains full precision, but because vector clock operations scale linearly in the number of threads, an analysis that performs many such operations will be slow.

Indirect Representation via Locksets. Analyses may trade off precision for speed by using an indirect approximation to the happens-before relation. For example, the lockset algorithm checks that conflicting accesses are guarded by a common lock. This is sufficient, but not necessary, for race freedom.

We describe all of these analyses in more detail below.

Lockset-based dynamic race detectors such as Eraser [17] keep track of the set of locks currently held by each thread and check that a consistent locking discipline

is used across threads. This approach depends on a locking synchronization discipline and is not easily extended to other synchronization idioms [17], though RaceTrack [20], discussed below, can be viewed as such an extension. Lockset approaches tend to report many false positives but can be implemented efficiently.

Happens-before-based dynamic race detectors such as DJIT⁺ and FASTTRACK use vector clocks to precisely track the happens-before relation [9, 7]. A vector clock is a data structure that stores a time for each thread in the program. Each thread typically keeps a vector clock containing the most recent value for each other threads' clock. Relative timing of operations in different threads can be determined by comparing vector clocks point-wise. Vector clocks are discussed in more detail in Section 2.2.

By using the happens-before relation explicitly, these detectors are easy to extend to other synchronization idioms. Vector clocks require space linear in the number of threads, and so operations on vector clocks are relatively expensive. Thus happens-before detectors must be carefully optimized to avoid vector clock operations wherever possible. For example, a Java implementation of DJIT⁺ required vector clock operations on 20-30% of accesses, while a comparable implementation of FASTTRACK reduces this to 0.1% of accesses. However, even FASTTRACK slowed a suite of benchmark programs down by a factor of 8 [7].

Hybrid Approaches. Some analyses combine multiple approaches. O'Callahan and Choi [15] mix the performance of lockset algorithms with the precision of happens-before detectors. Their tool initially uses a lockset algorithm as a fast but imprecise approximation that can detect which parts of the program should be analyzed more precisely (and at higher cost) using a happens-before detector. This approach is nearly as efficient as standard lockset analyses, while only missing a few races as compared to the happens-before detector it uses. These missed races stem from the two-phase approach. For example, if a program contains a single race that is caught by the lockset algorithm, it may not be reported because the happens-before tool would never see another race. MultiRace [16] is a similar technique that combines happens-before and lockset techniques.

RaceTrack [20] also leverages lockset and happens-before analysis to achieve performance and precision. In its most precise state, RaceTrack keeps a set of threads that are currently accessing each memory location. As this is expensive in general, it optimizes several common cases, including thread-local and read-shared data, which

were later exploited by FASTTRACK, as well. RaceTrack is neither sound nor complete. It is incomplete because it reports races to the user that were flagged by the lockset algorithm, which cannot handle non-locking synchronization idioms. It is unsound because it resets some locksets to empty when one thread believes it is the only thread to access that memory location. Using close integration with the VM and careful optimization, RaceTrack slows its target program down by a factor of only 1.3, while its memory overhead is only a factor of 1.2, and the authors reported that they were “unable to find comparative numbers in the literature.”

Vector Clocks and Happens-Before-Based Detection

Vector clocks allow relatively-efficient tracking of the happens-before relation. They are used by state-of-the-art precise dynamic race detectors, and are used as part of the main analysis studied in this work.

A vector clock is a vector of clocks. More formally, a vector clock $v \in VC$ is a map $Tid \rightarrow Nat$ from thread identifiers to the natural numbers, representing clock values for each thread. A vector clock captures the relative ordering of operations from different threads. This relative order is given by a partial order over vector clocks, defined by comparing each component. More precisely, given vector clocks v and w , we say $v \sqsubseteq w$ if $v(t) \leq w(t)$ for all thread ids t . This partial order has a corresponding join operation, which is given by $(v \sqcup w)(t) = \max(v(t), w(t))$. Finally, several operations need to advance a given entry in a vector clock, which is the purpose of the function inc_t .

$$\begin{aligned} inc_t &: VC \rightarrow VC \\ inc_t(v) &= v[t := v(t) + 1] \end{aligned}$$

The notation $v[t := k]$ represents the vector clock that is equal to v at all components except t , where it is equal to k . Thus $inc_t(v)$ increments the t component of v .

Example 2.6 (Vector Clocks). In case of two threads, we can write vector clocks as ordered pairs of numbers. So consider the clocks $[1, 0]$ and $[0, 1]$. These clocks are not ordered since they are not ordered component-wise. The join of these clocks is just $[1, 1]$. Now consider the clocks $[1, 0]$ and $[1, 1]$. We have $[1, 0] \sqsubseteq [1, 1]$ immediately, and $[1, 0] \sqcup [1, 1] = [1, 1]$.

$$\begin{array}{l}
t \in Tid \\
c \in Tid \rightarrow VC \\
l \in Lock \rightarrow VC \\
x \in Mem \\
r, w \in Mem \rightarrow VC
\end{array}$$

$$\begin{array}{c}
\text{[VC-ACQUIRE]} \\
\frac{c' = c[t := c_t \sqcup l_m]}{(c, l, r, w) \rightarrow^{acq(t,m)} (c', l, r, w)} \\
\\
\text{[VC-RELEASE]} \\
\frac{l' = l[m := c_t] \quad c' = c[t := inc_t(c_t)]}{(c, l, r, w) \rightarrow^{rel(t,m)} (c', l', r, w)} \\
\\
\text{[VC-READ]} \\
\frac{w_x \sqsubseteq c_t \quad r' = r[x := r_x[t := c_t(t)]]}{(c, l, r, w) \rightarrow^{rd(t,x)} (c, l, r', w)} \\
\\
\text{[VC-WRITE]} \\
\frac{r_x \sqsubseteq c_t \quad w_x \sqsubseteq c_t \quad w' = w[x := w_x[t := c_t(t)]]}{(c, l, r, w) \rightarrow^{wr(t,x)} (c, l, r, w')}
\end{array}$$

Figure 2.9: Formal rules for a simple race detector.

A Simple Race Detector. We now describe a relatively straightforward approach to race detection using vector clocks to track the happens-before relation. This analysis is summarized in Figure 2.9.

The analysis keeps a vector clock c_t for each thread t , a vector clock l_m for each lock m , and two clocks r_x and w_x for each memory location x . The t component of thread t 's clock, $c_t(t)$, is a local counter of time, while the other components of c_t record the last-observed values of the local counter of every other thread. The clock l_m records the clock of the last thread to release m . The t component of the clocks r_x and w_x record the last time that thread t performed a read or write to location x , respectively.

The clocks are updated as follows.

[VC-ACQUIRE]. Whenever thread t acquires a lock m , the clock c_t is set to $c_t \sqcup l_m$, which reflects the guarantee that this acquire happens after the most recent release of m .

[VC-RELEASE]. When t releases m , the clock l_m is set to c_t and then $c_t(t)$ is incremented. The first part ensures that any future acquire of m will happen after this release, while the second part guarantees that any access performed by t after this release will be correctly detected as racing with accesses by other threads to the same locations, even if those threads acquire m .

[VC-READ]. Whenever a thread t reads a location x , it verifies that $w_x \sqsubseteq c_t$ and sets $r_x(t)$ to $c_t(t)$. In other words, the read is race free if for every thread u , t knows at least as recent a value for u 's clock as the location x does. Phrased yet another way, the read is race free if no other thread has written to x after last communicating with thread t .

[VC-WRITE]. Whenever a thread t writes a location x , it checks $w_x \sqsubseteq c_t$ and $r_x \sqsubseteq c_t$, and then sets $w_x(t)$ to $c_t(t)$. This is similar to the case for reads, except that both reads and writes by other threads must be ordered with respect to the current clock for this thread.

Example 2.7 (Vector Clock State for a Correct Program). We now compute the state of the vector clocks above for trace A in Figure 2.5 at a few important program points. We'll continue to write clocks as ordered pairs since there are two threads: the first component represents the value of the clock on thread 1, and the second for thread 2. Our computations are summarized in Figure 2.10.

There are several clocks to keep track of: c_1 and c_2 represent the knowledge of threads 1 and 2 about their time relative to the other; l_m stores the clock of the most recent thread to release it; and r_x and w_x store the clocks of the most recent threads to read and write to x .

Initially, all clocks are $[0, 0]$, except $c_1 = [1, 0]$ and $c_2 = [0, 1]$. In trace A, thread 1 wins the race to acquire the lock first, so we set $c_1 = [1, 0] \sqcup [0, 0] = [1, 0]$. Nothing changes because thread 1 is the first thread to synchronize on lock m . Then thread 1 reads and writes x and finally releases m . Thus we have $r_x = [1, 0] = w_x$, $l_m = [1, 0]$ and $c_1 = [2, 0]$. Next, thread 2 acquires m , so $c_2 = [1, 1]$. Then thread 2 processes x and releases m , after which $r_x = [1, 1] = w_x$, $l_m = [1, 1]$, and $c_2 = [1, 2]$. This completes the trace.

Example 2.8 (Correct Program is Race free). Using the table computed at the end of Example 2.7, we now verify that all accesses in the execution are race free,

c_1	c_2	r_x	w_x	l_m
[1, 0]	[0, 1]	[0, 0]	[0, 0]	[0, 0]
↓ $\text{acq}(1, m)$				
[1, 0]	[0, 1]	[0, 0]	[0, 0]	[0, 0]
↓ $\text{rd}(1, x)$				
[1, 0]	[0, 1]	[1, 0]	[0, 0]	[0, 0]
↓ $\text{wr}(1, x)$				
[1, 0]	[0, 1]	[1, 0]	[1, 0]	[0, 0]
↓ $\text{rel}(1, m)$				
[2, 0]	[0, 1]	[1, 0]	[1, 0]	[1, 0]
<div style="display: flex; align-items: center; justify-content: center;"> <div style="margin-right: 10px;"> \swarrow $\text{acq}(2, m)$ </div> <div style="text-align: center;"> \downarrow </div> </div>				
[2, 0]	[1, 1]	[1, 0]	[1, 0]	[1, 0]
<div style="display: flex; align-items: center; justify-content: center;"> <div style="margin-right: 10px;"> \downarrow $\text{rd}(2, x)$ </div> <div style="text-align: center;"> \downarrow </div> </div>				
[2, 0]	[1, 1]	[1, 1]	[1, 0]	[1, 0]
<div style="display: flex; align-items: center; justify-content: center;"> <div style="margin-right: 10px;"> \downarrow $\text{wr}(2, x)$ </div> <div style="text-align: center;"> \downarrow </div> </div>				
[2, 0]	[1, 1]	[1, 1]	[1, 1]	[1, 0]
<div style="display: flex; align-items: center; justify-content: center;"> <div style="margin-right: 10px;"> \downarrow $\text{rel}(2, m)$ </div> <div style="text-align: center;"> \downarrow </div> </div>				
[2, 0]	[1, 2]	[1, 1]	[1, 1]	[1, 1]

Figure 2.10: Illustration of simple race detector on Trace A from Figure 2.5.

c_1	c_2	r_x	w_x	l_m
[1, 0]	[0, 1]	[0, 0]	[0, 0]	[0, 0]
\downarrow rd(1, x)				
[1, 0]	[0, 1]	[1, 0]	[0, 0]	[0, 0]
\downarrow wr(1, x)				
[1, 0]	[0, 1]	[1, 0]	[1, 0]	[0, 0]
\downarrow rd(2, x)				
—	$w_x \not\sqsubseteq c_2$	—	—	—

Figure 2.11: Illustration of simple race detector on the program from Example 2.1, using Trace A from Figure 2.2. A race is detected on the operation $\text{rd}(2, x)$ because $w_x \not\sqsubseteq c_2$.

using the above checks. Just before the read by thread 1, we see that $w_x = [0, 0] \sqsubseteq [1, 0] = c_1$, and so the read is race-free. Next, for the write by thread 1, we have $r_x = [1, 0] \sqsubseteq [1, 0] = c_1$ and $w_x = [0, 0] \sqsubseteq [1, 0]$ as desired. More interestingly, for the read by thread 2, we have $w_x = [1, 0] \sqsubseteq [1, 1] = c_2$. For the final write, we have $r_x = [1, 1] \sqsubseteq [1, 1] = c_2$ and $w_x = [1, 0] \sqsubseteq [1, 1] = c_2$. Thus all accesses in the trace were race free.

Example 2.9 (Program with Races). Going all the way back to Examples 2.1 to 2.3, we now verify that the above analysis catches the races. Our calculations are summarized in Figure 2.11. Since neither thread performs any synchronization operations, $c_1 = [1, 0]$ and $c_2 = [0, 1]$ for the whole program, and we need only worry about w_x and r_x . Consider trace A from Example 2.2. After thread 1 reads and writes to x , we have $r_x = [1, 0] = w_x$. When thread 2 reads from x , we check if $w_x \sqsubseteq c_2$, but $[1, 0] \not\sqsubseteq [0, 1]$ as we saw in Example 2.6, so the race is caught.

Efficient Vector-clock Techniques. Both DJIT⁺ and FASTTRACK improve on the straightforward approach sketched above by eliminating vector clock operations where possible. DJIT⁺ takes advantage of the fact that a thread’s clock $c_t(t)$ does not change unless that thread performs a synchronization operation. Thus, if a thread t accesses location x that it has already accessed since performing its last synchronization operation, DJIT⁺ avoids the vector clock check. Any race on the second

access would also be a race on the first. This optimization can be implemented for read operations by checking if $w_x(t) = c_t(t)$ before performing the full vector clock comparison $w_x \sqsubseteq c_t$. FASTTRACK goes further, exploiting the insight that, for most memory locations, all accesses are totally ordered. In this case, the full generality of a vector clock is not necessary, and FASTTRACK only stores the identity and clock of the last accessing thread. For example, in the case of writes, FASTTRACK completely eliminates the vector clock w_x and replaces it by the pair (t, e) where t is the last thread to write to x and e was the clock $c_t(t)$ at the time of the write. A similar simplification is performed on reads. Races can be detected on an access by thread u by only checking $e \leq c_u(t)$.

2.3 Race Detection on Arrays

The simple race detector presented in the previous section handles the operations listed in Table 2.1. In particular, all reads and writes have been to shared (scalar) variables, but we now consider how to extend the analysis to shared array variables. To this end, we consider the following additional operations.

`a_rd(t, a, i)` thread `t` reads index `i` of array `a`
`a_wr(t, a, i)` thread `t` writes index `i` of array `a`

We now consider how to extend the analysis to check these operations.⁴ The most natural approach is to handle each element of the array as a separate memory location, and apply the analysis “pointwise.” In other words, for each array a , and in-bounds index i , the analysis keeps two arrays of vector clocks, $w_a[i]$ and $r_a[i]$. These are the shadow state for the analysis, and we refer to them as the shadow arrays. Every read or write to an array element is checked using these vector clocks, as described by the scalar read and write rules. We refer to this technique of keeping one vector clock per array element as the fine-grained representation, which we abbreviate to fine mode.

To see how this analysis works on a simple array program, consider Figure 2.12. Here, each thread of the program acquires a common lock `m`, then touches every element of the array `a`, and then releases `m`. We show the state kept by the simple race detector, extended to handle array accesses. The vector clocks c_1 , c_2 , and l_m

⁴We describe our work for the simple vector clock algorithm, in order to avoid the complexities of FASTTRACK. Our techniques are equally applicable to that algorithm, and we discuss some of the more interesting details of the extension to FASTTRACK in Chapter 5.

Thread 1	Thread 2	a	c_1	c_2	l_m	w_a	w_a^{opt}
\vdots		$\boxed{0\ 0\ 0\ 0}$	[10,0]	[0,30]	[0,0]	$\boxed{[0,0]}\ \boxed{[0,0]}\ \boxed{[0,0]}\ \boxed{[0,0]}$	$\boxed{[0,0]}$
\downarrow							
a[0] = 0		$\boxed{0\ 0\ 0\ 0}$	[10,0]	[0,30]	[0,0]	$\boxed{[10,0]}\ \boxed{[0,0]}\ \boxed{[0,0]}\ \boxed{[0,0]}$	$\boxed{[0,0]}$
\downarrow							
a[1] = 1		$\boxed{0\ 1\ 0\ 0}$	[10,0]	[0,30]	[0,0]	$\boxed{[10,0]}\ \boxed{[10,0]}\ \boxed{[0,0]}\ \boxed{[0,0]}$	$\boxed{[0,0]}$
\downarrow							
a[2] = 2		$\boxed{0\ 1\ 2\ 0}$	[10,0]	[0,30]	[0,0]	$\boxed{[10,0]}\ \boxed{[10,0]}\ \boxed{[10,0]}\ \boxed{[0,0]}$	$\boxed{[0,0]}$
\downarrow							
a[3] = 3		$\boxed{0\ 1\ 2\ 3}$	[10,0]	[0,30]	[0,0]	$\boxed{[10,0]}\ \boxed{[10,0]}\ \boxed{[10,0]}\ \boxed{[10,0]}$	$\boxed{[10,0]}$
\downarrow							
rel m		$\boxed{0\ 1\ 2\ 3}$	[11,0]	[0,30]	[10,0]	$\boxed{[10,0]}\ \boxed{[10,0]}\ \boxed{[10,0]}\ \boxed{[10,0]}$	$\boxed{[10,0]}$
	acq m	$\boxed{0\ 1\ 2\ 3}$	[11,0]	[10,30]	[10,0]	$\boxed{[10,0]}\ \boxed{[10,0]}\ \boxed{[10,0]}\ \boxed{[10,0]}$	$\boxed{[10,0]}$
	\downarrow						
	a[0] = 4	$\boxed{4\ 1\ 2\ 3}$	[11,0]	[10,30]	[10,0]	$\boxed{[10,30]}\ \boxed{[10,0]}\ \boxed{[10,0]}\ \boxed{[10,0]}$	$\boxed{[10,0]}$
	\downarrow						
	a[1] = 5	$\boxed{4\ 5\ 2\ 3}$	[11,0]	[10,30]	[10,0]	$\boxed{[10,30]}\ \boxed{[10,30]}\ \boxed{[10,0]}\ \boxed{[10,0]}$	$\boxed{[10,0]}$
	\downarrow						
	a[2] = 6	$\boxed{4\ 5\ 6\ 3}$	[11,0]	[10,30]	[10,0]	$\boxed{[10,30]}\ \boxed{[10,30]}\ \boxed{[10,30]}\ \boxed{[10,0]}$	$\boxed{[10,0]}$
	\downarrow						
	a[3] = 7	$\boxed{4\ 5\ 6\ 7}$	[11,0]	[10,30]	[10,0]	$\boxed{[10,30]}\ \boxed{[10,30]}\ \boxed{[10,30]}\ \boxed{[10,30]}$	$\boxed{[10,30]}$
	\downarrow						
	rel m	$\boxed{4\ 5\ 6\ 7}$	[11,0]	[10,31]	[10,30]	$\boxed{[10,30]}\ \boxed{[10,30]}\ \boxed{[10,30]}\ \boxed{[10,30]}$	$\boxed{[10,30]}$
	acq m	$\boxed{4\ 5\ 6\ 7}$	[11,30]	[10,31]	[10,30]	$\boxed{[10,30]}\ \boxed{[10,30]}\ \boxed{[10,30]}\ \boxed{[10,30]}$	$\boxed{[10,30]}$
	\downarrow						
	\vdots	$\boxed{4\ 5\ 6\ 7}$	[11,30]	[10,31]	[10,30]	$\boxed{[10,30]}\ \boxed{[10,30]}\ \boxed{[10,30]}\ \boxed{[10,30]}$	$\boxed{[10,30]}$

Figure 2.12: Example of a common idiom: Acquire a lock, touch the entire array, release the lock. The analysis verifies that the program is race free using “fine-grained” shadow representation, w_a , which keeps a vector clock for each element of the array. On the far right, we show the “coarse-grained” shadow representation. In this case, because each thread touches the entire array without doing synchronization, the analysis could have used the coarse representation and retained full precision.

carry the same meaning as before. For each element of the array, the analysis keeps a vector clock. The figure shows these clocks in the array w_a , which we refer to as the shadow array.

This program is race free, and the detector reports no races. However, the detector can only verify the fact that the program is race free by performing a (relatively expensive) race check on each array access. Since no synchronization operations occur during the access sequence of a particular thread, these operations repeatedly check equivalent shadow states. Thus the analysis suffers from redundant race check operations.

Furthermore, at some important points during the program, the shadow array shows some serious redundancy as well. In particular, after the first thread has finished accessing the array but before the second thread has begun, the shadow array consists of four equal vector clocks. Similarly, after the second array finishes its accesses, the shadow array is again four equal vector clocks. Thus the analysis suffers from redundancy in the shadow array representation.

There is a standard technique for mitigating this space redundancy, which we refer to as the coarse-grained representation, or coarse mode. Instead of keeping a vector clock in the shadow state for each element, the array as a whole is treated as one abstract location and given a single vector clock. Thus, fine and coarse mode differ in the way they associate vector clocks in the shadow state to target array elements. In particular, fine mode maps each element of the target program array to a distinct vector clock in the shadow state, while coarse mode maps all elements of the target array to a single vector clock in the shadow state.

In coarse mode, each access to the array is checked using this one vector clock and the scalar rules above. In the example program of Figure 2.12, a coarse mode detector will correctly report no races, while using only a single vector clock, successfully eliminating the space redundancy. The redundant clock operations can now be eliminated as well, since all the checks are to a single clock. To do this, we simply note that the thread performs no synchronization operations while accessing the various array elements, and thus the thread's vector clock will be constant across all four clock operations. Since all four clock operations are to the same clock, it follows that the operations may be safely eliminated. This finishes the elimination of both types of redundancy from this simple program.

Thread 1	Thread 2	a	c_1	c_2	b	w'_a	w_a^{block}				
⋮	⋮	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	0	[10,0]	[0,30]	[0,0]	[10,0]	[10,0] [10,0]
0	0	0	0								
↓	↓										
bar b ← → bar b		<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	0	[11,30]	[10,31]	[10,30]	[10,0]	[10,0] [10,0]
0	0	0	0								
↓	↓										
a[0] = 4	a[2] = 6	<table border="1"><tr><td>4</td><td>0</td><td>6</td><td>0</td></tr></table>	4	0	6	0	[11,30]	[10,31]	[10,30]	!!!	[11,30] [10,31]
4	0	6	0								
↓	↓										
a[1] = 5	a[3] = 7	<table border="1"><tr><td>4</td><td>5</td><td>6</td><td>7</td></tr></table>	4	5	6	7	[11,30]	[10,31]	[10,30]	—	[11,30] [10,31]
4	5	6	7								
↓	↓										
bar b ← → bar b		<table border="1"><tr><td>4</td><td>5</td><td>6</td><td>7</td></tr></table>	4	5	6	7	[12,31]	[11,32]	[11,31]	—	[11,30] [10,31]
4	5	6	7								
↓	↓										
⋮	⋮	<table border="1"><tr><td>4</td><td>5</td><td>6</td><td>7</td></tr></table>	4	5	6	7	[12,31]	[11,32]	[11,31]	—	[11,30] [10,31]
4	5	6	7								

Figure 2.13: Example of another idiom: Worker threads divide an array into disjoint parts and perform work without synchronization. w'_a represents the shadow state for a detector that keeps only one vector clock for the whole array. This algorithm reports a race where there is none. w_a^{block} represents the ideal shadow state for this program: one vector clock per thread. This algorithm correctly verifies race freedom while saving space over the naïve fine mode.

However, there is a problem with this technique. In general, it is not precise to check accesses to an array using only a single vector clock for the entire array. This is because different elements of the array may be accessed by different threads without synchronization, and this is not a race. By using only a single vector clock, the detector will incorrectly report a race in such a case.

To understand this situation, consider a program with a more complicated access sequence. In Figure 2.13, the array is initialized by the first thread (not shown). A memory barrier then occurs. Briefly, the semantics of the barrier is to compute and distribute the component-wise maximum of all clocks, and then increment the local clock.

Next, the threads divide the array into two blocks of two indices each and do some work. Another barrier follows. Finally, the first thread reads every value of the array (not shown). Note that the program is race free since the threads access disjoint sets of indices. We have depicted the accesses in the threads as running in parallel instead of considering an interleaving, because we wish to discuss several possible interleavings.

```

void crypt(byte[] key, byte[] plain, byte[] cipher) {
    for (int i = 0; i < 4; i++) {
        // fork thread with id = i to execute run()
    }
}

void run(int id, byte[] key, byte[] plain, byte[] cipher) {
    int block = plain.length / 4;
    for (int i = id * block; i < (id + 1)*block; i++) {
        cipher[i] = mix(key, plain[i]);
    }
}

```

Figure 2.14: High-level code for the `crypt` benchmark.

The coarse mode optimization described above incorrectly reports a race on this program. First consider an interleaving where the thread 1 accesses index 0 before thread 2 accesses index 2. In this case, w'_a will be checked against [11,30], and the check will succeed, placing [11,30] in w'_a . At some later time in this interleaving, thread 2 will access index 2, and w'_a will have entries at least 11 and 30 respectively. But c_2 will be [10,31], which is not comparable. Thus a race will be reported. The case where thread 2 goes first is symmetric.

However, there is another, more sophisticated optimization to w_a that is precise in this situation. It is depicted as w_a^{block} in the right-most column of Figure 2.13. Here, the analysis treats the first two indices as a single abstract location and the last two indices as another abstract location. It thus keeps two vector clocks. Because the threads each only touch a single abstract location, and because they access all the constituent indices before performing synchronization, the analysis is able to correctly conclude that the program is race free.

The optimization of the previous paragraph is the motivation for our work. For example, consider the benchmark `crypt` that uses the above idiom of dividing an array into blocks and then working independently. Previous work induced a higher-than-average slow-down on this benchmark, and we sought to improve this performance. At a high level, `crypt` implements an algorithm such as the one in Figure 2.14. The procedure `mix` is not shown, but encapsulates the actual computation performed at index `i`.

Thread 1	Thread 2	a	c_1	c_2	b	w_a^{stride}						
\vdots	\vdots	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	0	[10,0]	[0,30]	[0,0]	<table border="1"><tr><td>[10,0]</td><td>[10,0]</td></tr></table>	[10,0]	[10,0]
0	0	0	0									
[10,0]	[10,0]											
\downarrow	\downarrow											
bar b	\leftrightarrow bar b	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	0	[11,30]	[10,31]	[10,30]	<table border="1"><tr><td>[10,0]</td><td>[10,0]</td></tr></table>	[10,0]	[10,0]
0	0	0	0									
[10,0]	[10,0]											
\downarrow	\downarrow											
a[0] = 4	a[1] = 6	<table border="1"><tr><td>4</td><td>6</td><td>0</td><td>0</td></tr></table>	4	6	0	0	[11,30]	[10,31]	[10,30]	<table border="1"><tr><td>[11,30]</td><td>[10,31]</td></tr></table>	[11,30]	[10,31]
4	6	0	0									
[11,30]	[10,31]											
\downarrow	\downarrow											
a[2] = 5	a[3] = 7	<table border="1"><tr><td>4</td><td>6</td><td>5</td><td>7</td></tr></table>	4	6	5	7	[11,30]	[10,31]	[10,30]	<table border="1"><tr><td>[11,30]</td><td>[10,31]</td></tr></table>	[11,30]	[10,31]
4	6	5	7									
[11,30]	[10,31]											
\downarrow	\downarrow											
bar b	\leftrightarrow bar b	<table border="1"><tr><td>4</td><td>6</td><td>5</td><td>7</td></tr></table>	4	6	5	7	[12,31]	[11,32]	[11,31]	<table border="1"><tr><td>[11,30]</td><td>[10,31]</td></tr></table>	[11,30]	[10,31]
4	6	5	7									
[11,30]	[10,31]											
\downarrow	\downarrow											
\vdots	\vdots	<table border="1"><tr><td>4</td><td>6</td><td>5</td><td>7</td></tr></table>	4	6	5	7	[12,31]	[11,32]	[11,31]	<table border="1"><tr><td>[11,30]</td><td>[10,31]</td></tr></table>	[11,30]	[10,31]
4	6	5	7									
[11,30]	[10,31]											

Figure 2.15: Example of a more complicated access pattern: strided access.

We have seen that in general the coarse mode optimization is not precise. However, the optimization is sound in the sense that if no races are reported then the program is race free. Thus, previous work has suggested first attempting to check the program using coarse mode and then falling back to fine mode if any races are reported. Indeed, MultiRace uses a technique that successively approximates fine mode. It begins with coarse mode and, if that fails (i.e., reports races), divides the array into two blocks with one vector clock for each block and checks the program again. This continues until no races are reported or until the array is represented in fine mode, so that all reported races are actual races [16].

This approach has the advantage that it can retain the speedups and memory reductions associated with coarse mode while also providing a completeness guarantee (that is, all reported races are real races). However, to achieve this completeness guarantee, one must wait for the program to be checked many times until the race is confirmed in fine mode. Thus it is possible to squander any time saved in coarse mode by repeatedly checking a program with a real race. Furthermore, this technique cannot help an “always on” detector that monitors running systems, since the analysis requires multiple runs of the program on the same input. Ideally, an analysis would begin in coarse mode until it was detected to be imprecise, at which point the analysis would transition to a more expensive but more precise mode and continue. Our analysis, described in Chapters 3 and 4, will do this.

Furthermore, the technique of repeated runs with finer representation is only beneficial on programs whose access patterns are blocks. However, many array programs use more complicated patterns. Consider, for example, the program of Figure 2.15, which is very similar to the previous program. The only difference is in how the two worker threads split up the array. Now the first thread accesses even indices while the second thread accesses the odd indices. We do not show the attempted coarse mode analysis; instead, we have shown only the ideal, two clock state. The abstract locations have changed to reflect the new access pattern, so the first clock represents even indices, and the second clock odd indices. This access pattern is not handled by the multiple passes of MultiRace, and, to our knowledge, no existing dynamic race detector is able to perform this optimization. Since non-blocked patterns are actually fairly common in practice, it is desirable to come up with more general techniques.

Hence, we have two challenges:

- Design an analysis that begins in an efficient but potentially imprecise mode and detects when that mode may lead to imprecision on the fly, backing off to a less efficient but precise representation. The analysis should be sound and complete.
- Describe array access patterns other than blocks and make race detection efficient on programs with those patterns. More generally, develop the general theory of access patterns and their efficient representation.

We tackle these challenges in the remainder of the thesis.

Chapter 3

Summary of Analysis

This chapter introduces our analysis, answering the challenges set forth at the end of Chapter 2. This chapter is intended both as preparation for the formalism presented in Chapter 4, as well as a sufficiently detailed and self-contained account so as to allow the reader to skim the more technical chapter that follows.

3.1 Overview

The chapter is organized as follows.

In Section 3.2 we describe how to take advantage of patterned array accesses by eliminating redundant shadow states and race checks. We discuss several common patterns that we have identified in benchmarks and develop compression techniques that are suited to each such pattern. Additionally, we develop a general description of all possible compression techniques based on partitioning the array.

In Section 3.3 we discuss how to dynamically infer an access pattern for an execution trace in which we can only process a single index at a time, as they are accessed by the target program. One challenge introduced by this limitation is that accesses cannot be checked for races immediately if we hope to use compression. We deal with the challenge of efficiently and precisely summarizing accessed indices that have not yet been checked for races. We also discuss the problem of changing access patterns, as well as access patterns that are not captured by our modes. Finally, we discuss how to perform race checks on summarized sets of indices, and how to adapt the representation on the fly.

Lastly, Section 3.4 briefly presents our performance expectations for the analysis. We summarize the redundancy eliminated by our approach and discuss the additional benefit due to relative lack of shared data structures. Finally, we discuss the fundamental constraint on any improvement to an optimized vector clock race detector: on average, the cost of processing an access must not exceed the cost of performing the standard race check.

3.2 Adaptive Array Representation

We have shown that the vector clock-based array race detector of Chapter 2 suffers from redundancy in several common cases. The coarse mode optimization proposed and briefly evaluated there can yield huge savings in both time and space, but it lacks the precision that we seek in our analysis. Furthermore, there are other, more subtle access patterns that produce redundancy in the race detector but that are not well-matched by coarse mode.

We begin this section with a review of the redundancy present in all these forms. Next, we pursue a careful analysis of the cases in which it is, in fact, precise to perform the coarse optimization. From this special case, we finally infer a general principle that constrains when a compression mode may be applied precisely.

When a shadow array is redundant. We are interested in two types of redundancy, viz shadow state redundancy and runtime check redundancy. These inefficiencies impact the space and time requirements of checking a program for races. More precisely, we are interested in common cases where the shadow array contains many equivalent vector clocks, and where the fine mode detector performs the same vector clock operations on the same clocks over and over. Chapter 2 discussed several cases of particular interest.

First, it is common for threads to access every element of an array without performing any synchronization operations between the accesses. In this case, every vector clock in the shadow array will be equivalent, and it follows that the coarse mode technique from the previous chapter will be precise.

Second, threads may partition the array into contiguous blocks such that different threads touch entire blocks without synchronization. We call this block mode. In this case, all the vector clocks of a particular block are equivalent. And, again, precision

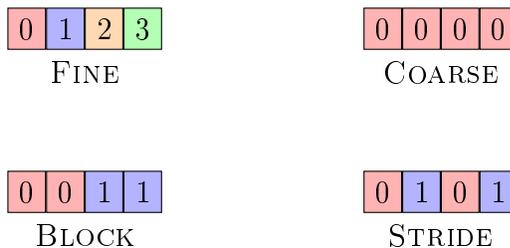


Figure 3.1: Four compression modes corresponding to common access patterns in array programs. Colors represent abstract locations. Whenever a thread accesses an element of a given color, it must access all other elements of that color before performing a synchronization operation. Fine handles arbitrary access patterns. Coarse handles programs that touch every element of the array without intervening synchronization. Block and Stride are two ways of partitioning the array.

is retained if we treat each block as one abstract location, with a single vector clock in the shadow state.

Finally, threads may access all indices separated by some fixed distance d from an initial index, which we call stride mode. Thus, the array is partitioned into equivalence classes modulo d , and the same redundancy discussion applies as above.

Figure 3.2 depicts each of the four modes we have considered so far: FINE, COARSE, BLOCK, and STRIDE.

When coarse mode is precise. Recall that the program of Figure 2.13 shows that coarse mode is not precise. A race detector running in coarse mode reports a race where there is none because it uses only a single vector clock to represent the entire array. Thus accesses to distinct elements of the array are unnecessarily required to be ordered by happens-before. We now ask the question: when could these unnecessary checks be harmless?

We first recall that coarse mode retained precision in the example of Figure 2.12. This is not hard to explain, since that program accesses the array all at once, in the sense that whenever a thread touches a single element of the array, it touches the rest of the array before performing a synchronization operation. This condition is sufficient for coarse mode to retain precision, since any two access sequences of this form race if and only if any single access from the first sequence is unordered with respect to any single access from the second.

Thus, we may explain the failure of coarse mode on the program of Figure 2.13 as

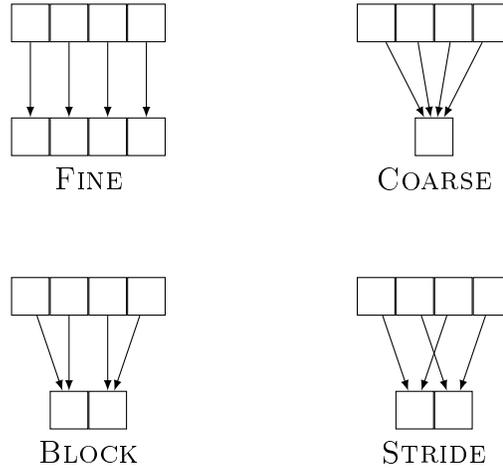


Figure 3.2: The function γ for each of the four modes.

a result of the more complicated access pattern. In particular, threads of this program touch only some elements of the array between the barriers, and these access sequences do not satisfy the condition above, and thus coarse mode need not be (in fact, is not) precise on this access sequence.

We may state the condition slightly more formally as follows.

Coarse Mode Principle, First Version. If, between synchronization operations, thread \mathbf{t} accesses each element of \mathbf{a} , then coarse mode may be used to precisely check \mathbf{a} for races.

Finally, we consider a rephrasing of the principle that will generalize more easily.

Coarse Mode Principle, Second Version. If, at every synchronization point, all the vector clocks in the fine representation of \mathbf{a} are equal, then coarse mode may be used to precisely check \mathbf{a} for races.

Other Modes. We now consider a generalization of this principle to other compression modes. For the purposes of this discussion, a compression mode is a partition of the valid indices of an array. The interpretation of such a mode is that it keeps one vector clock per equivalence class and performs all race checks for elements of the class relative to that class's vector clock. Numbering the equivalence classes $0, \dots, m-1$, we then specify the partition by a function $\gamma : \text{Nat} \rightarrow \text{Nat}$ such that $\gamma(i) = c$ if index i is in class c . We refer to regular indices into an array as concrete indices, and, for reasons that will become clear, to equivalence class numbers as abstract indices.

We now examine the partitions and γ functions for each of our four modes: FINE, COARSE, BLOCK, and STRIDE. See Figure 3.2.

FINE. FINE mode gives each index its own class. Thus $\gamma(i) = i$.

COARSE. COARSE mode lumps all indices into a single class, so $\gamma(i) = 0$.

BLOCK. BLOCK mode is parametrized by a length d , which is the number of consecutive indices that are grouped into a single class by the mode. If N is the length of the array, then there are N/d equivalence classes.¹ Numbering the equivalence classes in the natural way, γ sends indices $\{0, \dots, d-1\}$ to class 0, indices $\{d, 2d-1\}$ to class 1, and so on. Thus γ has a particularly nice representation, namely, $\gamma(i) = i \operatorname{div} d$, where div represents floored integer division.

STRIDE. STRIDE mode is also parametrized by a number d , which is the length of the skips between successive indices accessed in this pattern. The equivalence classes consist of all indices congruent to a fixed index modulo d . Thus γ again has a nice representation $\gamma(i) = i \bmod d$.

With these examples understood, we return to the problem of generalizing the above principle of precision.

Compression Principle, General Version. Suppose $S_{\mathbf{a}}$ is the fine mode shadow array for an array \mathbf{a} , and that $S'_{\mathbf{a}}$ is a proposed compression of $S_{\mathbf{a}}$, where the compression is specified by the function γ . Then whenever $S_{\mathbf{a}}(i) = S'_{\mathbf{a}}(\gamma(i))$ for all concrete indices i , $S'_{\mathbf{a}}$ precisely captures all information on $S_{\mathbf{a}}$.

Intuitively, this says that if, for each equivalence class, every element of the given class has the same shadow state, then the shadow array may be compressed by storing that unique shadow state only once for each class. Thus, we may use $S'_{\mathbf{a}}$ in place of $S_{\mathbf{a}}$ with no loss in precision.

3.3 Dynamic Mode Inference and Race Checks

The final version of our precise compression principle tells us that whenever the compressed state agrees with the uncompressed state, we can use the compressed state

¹For simplicity, we assume d divides N . The extension to the general case is handled in Chapter 4.

without loss of precision. However, after each access the fine mode shadow state is updated, and so any agreement that held between the compressed and uncompressed states is broken. For example, consider again the program from Figure 2.12. Just before thread 1 begins to access the array, the coarse mode compression precisely captures all the information in the fine mode array, because all the fine mode vector clocks are equal. However, after the first access, this is no longer true. Thus it is a nontrivial problem to actually use compressed shadow arrays to perform race detection, and it is this problem that we consider in this section.

We begin by discussing a precise method of postponing race detection that alleviates the concerns of the previous paragraph. Next, we describe a method for efficiently summarizing some sets of indices in such a way that corresponds nicely to the modes we considered in the previous section. Finally, we discuss how to actually perform the required race checks, taking into account that the access pattern may change on the fly.

Postponing race checks. We noted above that the invariant required for compression is immediately broken if we perform a race check on a particular index. This is due to the fact that any such race check updates the correct, fine mode shadow array in only one element, and thus breaks any invariant that depends on multiple elements. To fix this problem, we need a way of waiting until a thread has (hopefully) accessed many indices so that we can commit an entire equivalence class at a time. Of course, we want to wait as long as possible so as to maximize the advantages of the compression, but we cannot sacrifice precision.

Precision is not lost if we never allow an index to go unchecked across a synchronization operation. Again, this is due to the fact that a thread's vector clock does not change in the absence of synchronization operations. Thus, if an access from the current thread races with another access, as detected by a failed vector clock ordering operation, the accesses will still race if we wait until just before the next synchronization operation of the current thread to perform the check. The index's vector clock will only increase in each component, so if the current thread's clock does not happen after the last access to the index, it will also not happen after any subsequent access.

Summarizing accessed indices. Once we have decided to wait to check an index for races, we need a way of storing it so that it can be checked later. Placing the

index in a set data structure is one possibility, but the rather extreme time and space efficiency requirements under which we are working make even this option too expensive. Instead, we need an efficient way of summarizing the indices accessed since the last synchronization operation.

The method of summary should use constant space and support a constant time method for adding an index to the summary. Such a design is clearly impossible if one wishes to represent all possible index sets. Thus, we allow ourselves to leave some index sets unrepresentable.

Another design goal for index summary is that it should capture index sets that are likely to be seen in programs that also experience the type of redundancy we are trying to eliminate. In other words, we should try to represent as many equivalence classes of the above modes as possible.

Given these goals, a natural design is a *Footprint* $\langle b:e:k \rangle$, where b , e , and k are natural numbers: b , for beginning, represents the smallest index contained in the summary; e , for end, represents the largest index in the summary; finally, k is the distance between successive elements of the summary. Thus if $k = 1$, then the summary consists of all indices between b and e , inclusive. If $k = 2$, then the summary consists of b and then every other index, and so on.

For example, consider the program from Figure 2.12 again, which is reproduced in Figure 3.3. The footprints for each thread are shown in the two rightmost columns. At synchronization operations, the indices that have been summarized are checked for races (the vector clocks used for this are not shown). See the next chapter for more details and discussion.

With our summary in hand, our strategy is as follows: Record indices for a thread until either a synchronization operation occurs, or the index set becomes representable. Then perform race checks on all the summarized indices, and begin to record indices again, starting with the empty set.

Committing footprints. When it is time to perform race checks on all the indices summarized in a footprint, we say that we are committing the footprint. Footprints were designed so that access patterns well-matched with the modes could be summarized. When this is the case, the footprint typically covers one or more equivalence classes of the mode, and the race checks can be performed on the vector clocks for those classes. However, we must also be able to handle footprints from different

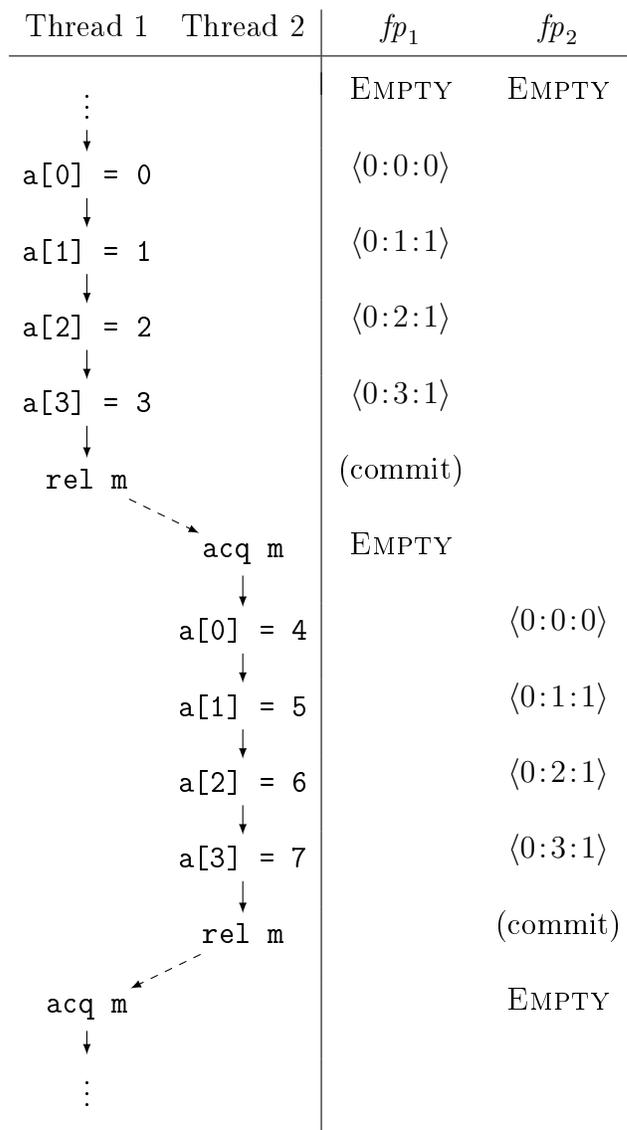


Figure 3.3: Postponing race checks using footprints to summarize indices accessed.

modes, or small footprints representing only a few indices, and not an entire equivalence class of any compressing mode. In this case, we will adapt the shadow array representation to a mode that does match the footprint being committed.

Note that every footprint covers a set of equivalence classes for fine mode (that is, no compression) since the equivalence classes consist of single indices. More generally, we will say that a footprint is compatible with a mode if the set of indices in the footprint is the union of some equivalence classes of the mode. Thus the above note can be rephrased as all footprints are compatible with fine mode.

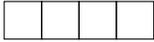
Previous Mode	Previous S_a	Footprint to Commit	Final Mode	Final S_a
COARSE		$\langle 0:3:1 \rangle$	(BLOCK 4)	
(BLOCK 4)		$\langle 0:1:1 \rangle$	(BLOCK 2)	
COARSE		$\langle 0:6:2 \rangle$	(STRIDE 2)	
(STRIDE 2)		$\langle 0:4:4 \rangle$	(STRIDE 4)	

Figure 3.4: Example transitions on an array of length 8.

We handle the problem of committing in two steps. First, we ensure that the footprint is compatible with the current mode by changing the current mode if necessary, as described below. Second, we discover which equivalence classes are covered by the footprint and perform the race checks on (and update) the corresponding vector clocks. After the first step is complete, compatibility of the footprint with the mode is assured, so that the footprint is guaranteed to cover some set of equivalence classes. Thus, all that remains is to describe how to ensure compatibility.

Ensuring compatibility of modes and footprints. Given a mode and a footprint that is incompatible with it, we want to find a mode that is compatible with the footprint such that it is efficient to transition from the current mode to the new mode. Furthermore, we should choose the new mode that compresses as much as possible. These constraints essentially determine a choice for each possible case of footprint and mode. See Figure 3.4 for some examples.

We now describe which transitions are considered efficient.

- We will say that it is efficient to transition to FINE from any mode, since we can make copies of the right shadow state for each element.
- It is also efficient to transition from block mode of length d to block mode of length d' if d' divides d . In this case, the blocks of the finer mode fit nicely inside the blocks of the coarser mode, and so we can construct the finer mode by copying, again.
- Finally, it is efficient to transition from stride mode of skip d to stride mode of skip d' if d divides d' (reverse of the previous condition). This is because a

larger skip d' corresponds to less compression, and because the divisibility of d' by d means that several small skips add up to a big skip.

All of these remarks may be summarized by saying that a mode transition is efficient if the destination mode is a refinement of the source mode, when both are viewed as partitions of the set of valid indices.

If any of the block-to-block or stride-to-stride transitions apply, we prioritize them over falling back to fine mode. In general, it is not possible to detect the most compressed mode without traversing the entire shadow array. Thus we settle for some heuristics that depend on the footprint being committed. These are described in more detail in the following chapter, but for now it suffices to say that in the block-to-block case, we choose the block of length $e - b$, if that quantity divides the current block length.

3.4 Expected Improvements

Our analysis eliminates redundancy that is encountered in practice. In particular, when threads partition an array and touch entire parts without performing synchronization, we seek to eliminate the repetitive vector clock operations and redundant states that result. We have identified two especially common access patterns, block and stride. For each mode, we have discussed how to compress the shadow array, how to postpone race checks by index set summary in footprints, and how to commit footprints while making a mode transition if necessary. Overall, this leads to savings in memory overheads due to fewer shadow states being needed, and savings in runtime overhead due to fewer race checks being performed.

There is an additional benefit that we have not discussed. Namely, that vector clocks are necessarily shared state and thus must be synchronized to ensure that the detector itself does not suffer from data races. This synchronization makes vector clock operations even more expensive than one might expect. On the other hand, footprints are entirely thread-local structures, and may be updated without synchronization. Synchronization is thus only necessary when committing footprints. This is a win if commits are relatively infrequent, as they will be in programs with access patterns well-matched by our modes.

More generally, in thinking about the performance of our analysis, it is important to keep in mind the following fundamental constraint if we hope to beat current dynamic race detectors.

The cost of an array access under our analysis must be faster on average than a standard race check.

This constraint is especially important when we compare ourselves to a state-of-the-art detector such as FASTTRACK, since FASTTRACK eliminates many expensive vector clock operations, replacing them with fast integer operations in the common case. Thus our analysis can only afford to average a few instructions per access. As we shall see in Chapter 5, we are able to do this efficiently much of the time, especially when almost all of the array accesses in the target program are involved in large patterns. However, we will also have cases (such as random access patterns) where footprints are less time-efficient than fine mode.

Chapter 4

Analysis

We now formalize and present the details of our analysis. We begin by formalizing a small language of operations and traces similar to those informally introduced in Chapter 2. Next, we formalize an unoptimized race detection algorithm as a semantics on traces. We then introduce footprints and compression modes, and compatibility of footprints with modes. We present our optimized race detector as a second semantics on traces. Finally, we formalize the relationship between the unoptimized and optimized detectors and prove that the two semantics are equivalent by proving a bisimulation theorem. We conclude by describing how to interpret this as the fact that our optimizations preserve soundness and completeness of the any underlying race detection algorithm.

4.1 Language

We define a simple language of operations performed by a multithreaded program. See Figure 4.1. *Tid* is the type of thread identifiers which uniquely name threads in the program. We will use the metavariable t to refer to thread identifiers. *Lock* is the type of lock names. Note that these names are run-time names, and so one may think of the lock name as its address in memory. This avoids issues of aliasing when reasoning about locks statically. We will use the metavariables l and m to refer to locks. The metavariables i and j refer to array indices.

We refer to single operations with the metavariable a . An operation is either an array access, a lock acquire, or a lock release. An operation $acc(t, i)$ denotes an access

$$\begin{aligned}
s, t &\in Tid \\
l, m &\in Lock \\
i, j &\in Nat \\
\\
a &\in Op ::= acc(t, i) \mid acq(t, l) \mid rel(t, l) \\
\alpha &\in Trace = Op^*
\end{aligned}$$

Figure 4.1: Simple operation language and traces. The analysis works on traces, which are interleavings of operations performed by the program.

to index i by thread t ; $acq(t, l)$ denotes a lock acquire on l by thread t ; finally, $rel(t, l)$ denotes a lock release on l by thread t .

We make several simplifying assumptions. First we assume there is a single array in the program, so that every index refers to the same array. We denote the length of this array by N . Second, we do not consider any memory accesses to non-array objects. Third, we do not distinguish reads and writes, simply labeling both as accesses. Finally, we do not consider synchronization operations other than lock acquire and release. All of these assumptions make the formalism cleaner, but of course our implementation deals with them appropriately. We will discuss the required extensions to the analysis when we present our implementation in Chapter 5.

A trace is simply a list of operations, which denotes an observed interleaving of operations performed by the threads. We assume traces are valid with respect to the synchronization primitives. Since the only synchronization operations we are considering are lock acquires and releases, validity just means that a thread cannot acquire a lock that a different thread already holds and it cannot release a lock that it does not hold.¹ Furthermore, we assume that a thread never re-acquires a lock that it already holds.² We also assume sequential consistency. That is, we assume that executions of multithreaded programs can be accurately represented by a trace. This assumption is standard within the race detection community when formalizing analyses.

¹What it means for a trace to be valid becomes more complicated when more synchronization primitives are considered, but even when we extend the formalism in Chapter 5 to handle all the Java synchronization primitives, we will not need to worry about trace validity, since traces produced by real Java programs are automatically valid.

²The extension to re-entrant locks is straightforward.

4.2 Unoptimized Detector Semantics

We now precisely describe the behavior of an unoptimized race detector. In Figure 4.2 we present a semantics describing the detector as well as the state required by the algorithm.

The basic detector we present here is essentially the algorithm discussed at the end of Chapter 2. We have chosen this algorithm because its simplicity makes the discussion clearer. The ideas and theorems carry over to any vector clock-based dynamic race detector.³ We review from our previous discussion some of the details of how this algorithm works.

As described in Chapter 2, a vector clock is a finite partial map from threads to clock values. Whenever we quantify over the arguments to a vector clock, we implicitly restrict ourselves to those values for which the map is defined. Since the map is finite, in all such cases there are at most a finite number of such thread identifiers. The ordering on vector clocks is the pointwise extension of the ordering on the natural numbers.

The basic algorithm keeps a vector clock \mathbb{C}_t for each thread t , which represents the maximum clock value that thread t has observed for each thread. Note that $\mathbb{C}_t(t)$ is thread t 's current clock, which gets incremented every time thread t participates in a release-like synchronization operation. The basic algorithm also keeps a vector clock \mathbb{L}_l for each lock l , which represents the maximum clock value over all threads that have ever released the lock. Finally, the basic algorithm keeps a vector clock \mathbb{W}_i for each index in the array, whose t component is the value of thread t 's clock at the last time t accessed index i . Recall that we are assuming a single shared array in the program, and no non-array accesses. These assumptions are reflected in the semantics, and this is essentially the only way in which our basic detector differs from that of Chapter 2.

Race detection in the basic algorithm is accomplished by keeping all the vector clocks updated in such a way that if two threads touch the same element of the array without intervening synchronization, the vector clock ordering requirement in the hypothesis of [BASIC ACC] will not hold. We now describe the function of each rule.

[BASIC ACC]. We have already mentioned that the comparison test in the hy-

³In fact, using vector clocks is not a requirement. Our ideas apply easily to heavily optimized happens-before-based detectors, such as FASTTRACK. The extension to more exotic dynamic race detectors would require some care, but we expect that most algorithms could be accommodated.

$$VC = Tid \rightarrow Nat$$

$$V \sqsubseteq W \Leftrightarrow \forall t. V(t) \leq W(t)$$

$$(V \sqcup W)(t) = \max(V(t), W(t))$$

$$\begin{aligned} inc_t &: VC \rightarrow VC \\ inc_t(V) &= V[t := V(t) + 1] \end{aligned}$$

$$\sigma \in BasicState = (\mathbb{C}, \mathbb{L}, \mathbb{W})$$

$$\begin{aligned} \mathbb{C} &: Tid \rightarrow VC \\ \mathbb{L} &: Lock \rightarrow VC \\ \mathbb{W} &: Nat \rightarrow VC \end{aligned}$$

$$\boxed{\sigma \rightarrow^a \sigma'}$$

[BASIC ACC]

$$\frac{\mathbb{W}_i \sqsubseteq \mathbb{C}_t \quad \mathbb{W}' = \mathbb{W}[i := \mathbb{W}_i[t := \mathbb{C}_t(t)]]}{(\mathbb{C}, \mathbb{L}, \mathbb{W}) \rightarrow^{acc(t,i)} (\mathbb{C}, \mathbb{L}, \mathbb{W}')}$$

[BASIC ACQ]

$$\frac{\mathbb{C}' = \mathbb{C}[t := \mathbb{C}_t \sqcup \mathbb{L}_t]}{(\mathbb{C}, \mathbb{L}, \mathbb{W}) \rightarrow^{acq(t,l)} (\mathbb{C}', \mathbb{L}, \mathbb{W})}$$

[BASIC REL]

$$\frac{\begin{aligned} \mathbb{L}' &= \mathbb{L}[l := \mathbb{C}_t] \\ \mathbb{C}' &= \mathbb{C}[t := inc_t(\mathbb{C}_t)] \end{aligned}}{(\mathbb{C}, \mathbb{L}, \mathbb{W}) \rightarrow^{rel(t,l)} (\mathbb{C}', \mathbb{L}', \mathbb{W})}$$

$$\boxed{\sigma \rightarrow^\alpha \sigma'}$$

[BASIC EMPTY]

$$\frac{}{\sigma \rightarrow^\epsilon \sigma}$$

[BASIC STEP]

$$\frac{\sigma \rightarrow^\alpha \sigma' \quad \sigma' \rightarrow^a \sigma''}{\sigma \rightarrow^{\alpha a} \sigma''}$$

Figure 4.2: One-step and trace semantics for an unoptimized race detector.

pothesis is what actually reports races. The other hypothesis updates the state at index i to reflect the fact that thread t touched it by setting the t component of the clock to the local clock of t , that is, $\mathbb{C}_t(t)$. If any other thread attempts to access index i without synchronizing with t , a race will thus be reported.

[BASIC ACQ]. The hypothesis updates t 's vector clock with the latest clock values that have been communicated through the lock.

[BASIC REL]. The first hypothesis updates the vector clock of l to reflect the latest values that have been learned by t and should be communicated to the next thread to acquire the lock.

[BASIC EMPTY] and [BASIC STEP]. We extend the above relation transitively to traces (lists) of operations.

This semantics detects races in traces in the following sense. If $\sigma_0 \rightarrow^\alpha \sigma$ then the trace α has no races. Here, σ_0 is the starting state of the detector, which is defined as

$$\sigma_0 = (\lambda t. inc_t(\mathbf{0}), \lambda l. \mathbf{0}, \lambda t. \mathbf{0}),$$

where $\mathbf{0} = \lambda s. 0 \in VC$ is the zero vector clock. Conversely, if there is no state σ such that $\sigma_0 \rightarrow^\alpha \sigma$, then the trace α contains a race.

4.3 Optimized Detector Semantics

$$\begin{aligned} b, e &\in Nat \\ k &\in \{1, 2, 3, \dots\} \\ fp &\in FootPrint ::= \langle b:e:k \rangle \\ \llbracket \cdot \rrbracket &: FootPrint \rightarrow 2^{Nat} \\ \llbracket \langle b:e:k \rangle \rrbracket &= \{i \mid b \leq i \leq e \text{ and } i \equiv b \pmod k\} \end{aligned}$$

Figure 4.3: Footprints and their interpretations.

$$\begin{aligned}
\oplus & : \text{FootPrint} \times \text{Nat} \rightarrow \text{FootPrint} \\
\text{EMPTY} \oplus i & = \langle i:i:1 \rangle \\
\langle b:b:1 \rangle \oplus (b+k) & = \langle b:(b+k):k \rangle \\
\langle b:b:1 \rangle \oplus (b-k) & = \langle (b-k):b:k \rangle \\
\langle b:e:k \rangle \oplus i & = \langle b:e:k \rangle \text{ if } b \leq i \leq e \text{ and } i \equiv b \pmod k \\
\langle b:e:k \rangle \oplus (e+k) & = \langle b:(e+k):k \rangle \\
\langle b:e:k \rangle \oplus (b-k) & = \langle (b-k):e:k \rangle
\end{aligned}$$

Figure 4.4: Extending a footprint by an index.

Footprints. Figure 4.3 introduces footprints and the sets of array indices they represent. The metavariables b , e , and k are taken to be natural numbers. Each also has an informal meaning. Since a footprint is simply a triple of natural numbers, we use b , e , and k always to refer to the components of the footprint. b represents the beginning index, e the ending index and k the stride. The metavariable fp refers to footprints.

The interpretation function $\llbracket \cdot \rrbracket$ describes the set of indices represented by a footprint. In particular, a footprint $\langle b:e:k \rangle$ represents all the indices between the endpoints (inclusive) that appear at integer multiples of the stride from the beginning.

A footprint is empty if and only if $e < b$, so there are many choices of b , e , and k such that $\llbracket \langle b:e:k \rangle \rrbracket = \emptyset$. We define a canonical empty footprint, $\text{EMPTY} = \langle 1:0:1 \rangle$. Similarly, if $k > 1$, it may be possible to represent the same set of indices with multiple footprints. For simplicity, we always assume that all empty footprints are equal to EMPTY , all singleton footprints are equal to $\langle i:i:1 \rangle$, and that $b \equiv e \pmod k$ for all footprints.

Extending Footprints. We now discuss how to use footprints to keep track of the indices accessed by a particular thread during a release-free span. Given a footprint fp and an index i , the operation $fp \oplus i$ attempts to construct a new footprint fp' that is the *extension* of fp by i , that is, such that $\llbracket fp' \rrbracket = \llbracket fp \rrbracket \cup \{i\}$. See Figure 4.4.

The result of $fp \oplus i$ is defined by considering several cases.

- If fp is empty, then the singleton footprint $\langle i:i:1 \rangle$, which represents exactly the index i , is the desired extension. This corresponds to the first line in the definition.

- If fp is itself a singleton, then there is no predetermined stride for the extension, so we may set it however we wish. We create a 2-element footprint stride $|b - i|$. This case is handled by lines 2 and 3 in the definition, depending on whether i is less than or greater than b . Note that \oplus always succeeds in extending an empty or singleton footprint by any index.
- Finally, there is the typical case, where fp has at least two elements. Here, the stride is predetermined, and so the extension succeeds only when the new index is already in the footprint or exactly one stride off of either end of the footprint. This matches the common case of programs that access arrays left-to-right or right-to-left. These cases are handled by the last three lines in the definition.

The function \oplus is undefined if none of the above cases match. Furthermore, \oplus is conservative in the sense that it only constructs certain classes of extensions, namely adding a new largest or smallest element. Thus there exist footprints and indices such that \oplus will not aggregate them, even though an extension exists.⁴

Consider the following examples of footprints being extended.

fp	i	$fp \oplus i$
EMPTY	0	$\langle 0:0:1 \rangle$
$\langle 0:0:1 \rangle$	1	$\langle 0:1:1 \rangle$
$\langle 0:1:1 \rangle$	2	$\langle 0:2:1 \rangle$
$\langle 0:0:1 \rangle$	4	$\langle 0:4:4 \rangle$
$\langle 0:4:4 \rangle$	8	$\langle 0:8:4 \rangle$

In each case we show the initial footprint and the index to be aggregated, followed by the resulting footprint.

Modes. We now formalize several ways of compressing redundant array shadow states so that it is efficient to perform lookups using the original index. Recall that we are assuming that there is only one array in the program, and its length is denoted by N . A mode M is either (BLOCK d) or (STRIDE d), where $d \in \{1, 2, 3, \dots\}$. In mode (BLOCK d), the uncompressed array is conceptually treated as a sequence of

⁴Consider $fp = \langle 0:4:4 \rangle$ and $i = 2$. Then $\llbracket \langle 0:4:2 \rangle \rrbracket = \llbracket fp \rrbracket \cup \{i\}$, but \oplus never modifies the stride of a non-singleton footprint, and so it will not find this extension. In fact, this is essentially the only case where an extension footprint exists but \oplus won't find it. It would be trivial to add this case to \oplus , but we have not found it necessary, and it is important to keep \oplus as simple and efficiently implementable as possible, since it is used on every array access.

$$\begin{aligned}
& d \in \text{Nat} \\
& M \in \text{Mode} ::= (\text{BLOCK } d) \mid (\text{STRIDE } d) \\
& \gamma : \text{Mode} \times \text{Nat} \rightarrow \text{Nat} \\
& \gamma((\text{BLOCK } d), i) = i \operatorname{div} d \\
& \gamma((\text{STRIDE } d), i) = i \operatorname{mod} d
\end{aligned}$$

Figure 4.5: Modes describe how to compress an array by specifying the size of the compressed array, and a function mapping uncompressed indices to compressed indices.

blocks of length d . Each element of a block-compressed array corresponds to an entire block of elements from the uncompressed array. If the array length is not divisible by d , then the last block is truncated. We are motivated to consider the case where d need not divide N because it arises frequently in practice. In mode $(\text{STRIDE } d)$, the array is divided into d interleaved sequences, where within each sequence the difference between successive indices is d . These modes capture common access patterns observed in programs. For example, $(\text{BLOCK } d)$ can compress an entire row of a 2-dimensional array stored in row-major order, while $(\text{STRIDE } d)$ can compress a column. See Figure 4.5.

We define two abbreviations corresponding to modes introduced informally in previous chapters. First, $\text{FINE} = (\text{BLOCK } 1)$, where each index is in its own block. Next, $\text{COARSE} = (\text{BLOCK } N)$, so that the entire array is considered as a single block.

The function γ maps uncompressed indices to compressed indices. In other words, to access the index i in the original array, we access the index $\gamma(M, i)$ in the compressed array when the compression is done under mode M .

Compatibility of footprints and modes. Informally, we say that a footprint is compatible with a mode if it represents a set of indices that is the union of equivalence classes under the mode thought of as a partition. The relation $M \vDash fp$ formalizes this notion. The rule [FP COMPAT] exactly encodes the fact that indices in the same equivalence class induced by M must either both be in the footprint or neither be in the footprint.

$$\begin{array}{c}
\boxed{M \vDash fp} \\
\text{[FP COMPAT]} \\
\frac{\forall i, j \in 1..N. (\gamma(M, i) = \gamma(M, j)) \Rightarrow (i \in \llbracket fp \rrbracket \Leftrightarrow j \in \llbracket fp \rrbracket)}{M \vDash fp}
\end{array}$$

$$\begin{array}{c}
\boxed{M \vdash fp} \\
\text{[BLOCK ALGO COMPAT]} \\
\frac{b \equiv 0 \pmod{d} \quad e \equiv -1 \pmod{d} \text{ or } e = N - 1}{(\text{BLOCK } d) \vdash \langle b:e:1 \rangle}
\end{array}
\qquad
\begin{array}{c}
\text{[STRIDE ALGO COMPAT]} \\
\frac{b < d \quad e \equiv b \pmod{d} \quad N - d \leq e < N}{(\text{STRIDE } d) \vdash \langle b:e:d \rangle}
\end{array}$$

$$\rho : \text{Mode} \times \text{FootPrint} \rightarrow 2^{\text{Nat}}$$

$$\rho((\text{BLOCK } d), \langle b:e:k \rangle) = \{i \text{ div } d \mid i \in [b, e]\} \quad \text{if } (\text{BLOCK } d) \vdash \langle b:e:k \rangle$$

$$\rho((\text{STRIDE } d), \langle 0:N-1:1 \rangle) = \{0, 1, \dots, d-1\}$$

$$\rho((\text{STRIDE } d), \langle b:e:k \rangle) = \{b\} \quad \text{if } (\text{STRIDE } d) \vdash \langle b:e:k \rangle$$

Figure 4.6: Compatibility of modes and footprints.

$\sigma \in \text{BasicState} = (\mathbb{C}, \mathbb{L}, \mathbb{W})$ $\begin{aligned} \mathbb{C} &: \text{Tid} \rightarrow VC \\ \mathbb{L} &: \text{Lock} \rightarrow VC \\ \mathbb{W} &: \text{Nat} \rightarrow VC \end{aligned}$	$\tau \in \text{OptimizedState} = (C, L, F, M, V)$ $\begin{aligned} C &: \text{Tid} \rightarrow VC \\ L &: \text{Lock} \rightarrow VC \\ F &: \text{Tid} \rightarrow \text{FootPrint} \\ M &: \text{Mode} \\ V &: \text{Nat} \rightarrow VC \end{aligned}$
$\sigma \rightarrow^a \sigma'$	$\tau \rightsquigarrow^a \tau'$
<p>[BASIC ACC]</p> $\frac{\mathbb{W}_i \sqsubseteq \mathbb{C}_t \quad \mathbb{W}' = \mathbb{W}[i := \mathbb{W}_i[t := \mathbb{C}_t(t)]]}{(\mathbb{C}, \mathbb{L}, \mathbb{W}) \rightarrow^{\text{acc}(t,i)} (\mathbb{C}, \mathbb{L}, \mathbb{W}')}$	<p>[OPT ACC]</p> $\frac{F' = F[t := F(t) \oplus i]}{(C, L, F, M, V) \rightsquigarrow^{\text{acc}(t,i)} (C, L, F', M, V)}$
<p>[BASIC ACQ]</p> $\frac{\mathbb{C}' = \mathbb{C}[t := \mathbb{C}_t \sqcup \mathbb{L}_t]}{(\mathbb{C}, \mathbb{L}, \mathbb{W}) \rightarrow^{\text{acq}(t,l)} (\mathbb{C}', \mathbb{L}, \mathbb{W})}$	<p>[OPT ACQ]</p> $\frac{C' = C[t := C_t \sqcup L_t]}{(C, L, F, M, V) \rightsquigarrow^{\text{acq}(t,l)} (C', L, F, M, V)}$
<p>[BASIC REL]</p> $\frac{\mathbb{L}' = \mathbb{L}[l := \mathbb{C}_t] \quad \mathbb{C}' = \mathbb{C}[t := \text{inc}_t(\mathbb{C}_t)]}{(\mathbb{C}, \mathbb{L}, \mathbb{W}) \rightarrow^{\text{rel}(t,l)} (\mathbb{C}', \mathbb{L}', \mathbb{W})}$	<p>[OPT REL]</p> $\frac{F(t) = \text{EMPTY} \quad L' = L[l := C_t] \quad C' = C[t := \text{inc}_t(C_t)]}{(C, L, F, M, V) \rightsquigarrow^{\text{rel}(t,l)} (C', L', F, M, V)}$

Figure 4.7: One step semantics for basic and optimized race detectors.

The algorithmic compatibility relation $M \vdash fp$ provides a definition of compatibility that is more efficient to compute than the semantically defined $M \vDash fp$. The rules for algorithmic compatibility $M \vdash fp$ are designed to agree with the semantic compatibility relation $M \vDash fp$ in all cases.

Finally, the function ρ computes the set of abstract indices covered by a footprint in a given mode. The function ρ satisfies $\rho(M, fp) = \gamma(M, \llbracket fp \rrbracket)$.

Optimized Semantics We now precisely describe the behavior of the optimized analysis, as well as its relationship to the basic analysis. In Figure 4.7 we present the semantics for operations under the optimized analysis. We repeat the semantics of the basic detector for convenience.

$$\begin{array}{c}
\boxed{\tau \equiv \tau'} \\
\text{[EQUIV COMMIT]} \\
\frac{M \models F(t) \quad \forall i \in \rho(M, F(t)). V(i) \sqsubseteq C_t \quad F' = F[t := \text{EMPTY}]}{V' = V[i := V(i)[t := C_t(t)]]^{\forall i \in \rho(M, F(t))}} \\
\frac{}{(C, L, F, M, V) \equiv (C, L, F', M, V')} \\
\text{[EQUIV MODE]} \\
\frac{\forall i \in 1..N. V'(\gamma(M', i)) = V(\gamma(M, i))}{(C, L, F, M, V) \equiv (C, L, F, M', V')} \\
\text{[EQUIV TRANS]} \\
\frac{\tau \equiv \tau' \quad \tau' \equiv \tau''}{\tau \equiv \tau''}
\end{array}$$

Figure 4.8: Committing and change of mode.

The optimized detector is, in many respects, the same as the basic detector. We highlight the differences. First, instead of the vector clock \mathbb{W}_i for each index, we have a map $V : \text{Nat} \rightarrow VC$ and a mode M . This implements the idea that V is a compression of \mathbb{W} , as described by M . Additionally, the optimized algorithm keeps a footprint $F(t)$, for each thread t , that represents the indices thread t has accessed since its last release-like synchronization operation. These accesses have not yet been checked for race conditions.

On an access, the analysis attempts to aggregate the index into the footprint for the current thread. When the resulting index set is not representable, we have left \oplus undefined, so that the rule [OPT ACC] does not apply. This means that the only way for the analysis to proceed is to first perform a commit, which we describe below. The rule for acquiring a lock is identical to the basic case. On a lock release, the first hypothesis of the optimized rule requires that there be no indices still waiting to be checked by the current thread. This hypothesis can only be satisfied if there have been no accesses, or if we have just performed a commit operation.

Comitting footprints. The process of committing is captured by the relation $\tau \equiv \tau'$ on optimized states, which indicates when two optimized states are conceptually

equivalent. In essence, the relation indicates when a footprint of accesses can be committed. Consider the rule [EQUIV COMMIT]. The first hypothesis ensures that the footprint we are about to commit is compatible with the current mode. The second hypothesis performs all the race checks, using the function ρ to consider all the abstract indices covered by the footprint. The third hypothesis clears the footprint for the committing thread, and the last hypothesis updates the relevant vector clocks in the shadow array.

Of course, the footprint for thread t may not be compatible with the current mode. That is, the relation $M \models F(t)$ does not always hold.

Rule [EQUIV MODE] captures how to convert the array’s representation to a different mode, so that the commit may be applied. The hypothesis of [EQUIV MODE] encodes the fact that both modes faithfully compress the same shadow array. Note that we make no mention of whether it is possible to efficiently make this transition, which was discussed in Chapter 3. This lack of restriction makes it possible for an implementation to choose the transitions it wants to make, and all of the theorems below will still hold. Rule [EQUIV MODE] ensures that \equiv is reflexive, and [EQUIV TRANS] makes \equiv transitive.

The rules of Figure 4.9 extend the step relations above to traces (i.e., lists) of operations. Any state steps to itself on the empty trace. In the basic transitive rule, any trace may be extended by simply stepping by one operation on the right. The optimized transitive rule is more complicated. Intuitively, it allows an intermediate mode change or commit before processing the next operation. The optimized analysis begins in state τ_0 .

Figure 4.10 relates basic and optimized states. In particular, an optimized state is equivalent to a basic state if their vector clocks agree for all indices of the array, after the optimized state has committed. Note that $\sigma_0 \sim \tau_0$.

4.4 Equivalence of the Two Semantics

We will now prove that the optimized semantics is equivalent to the original semantics by proving a bisimulation theorem relating the two. Informally, the semantics detect race conditions by “getting stuck.” That is, the trace contains a race if at some

$\boxed{\sigma \rightarrow^\alpha \sigma'}$ <p>[BASIC EMPTY]</p> $\frac{}{\sigma \rightarrow^\epsilon \sigma}$ <p>[BASIC STEP]</p> $\frac{\sigma \rightarrow^\alpha \sigma' \quad \sigma' \rightarrow^a \sigma''}{\sigma \rightarrow^{\alpha a} \sigma''}$	$\boxed{\tau \rightsquigarrow^\alpha \tau'}$ <p>[OPT EMPTY]</p> $\frac{}{\tau \rightsquigarrow^\epsilon \tau}$ <p>[OPT STEP]</p> $\frac{\tau \rightsquigarrow^\alpha \tau' \quad \tau'' \rightsquigarrow^a \tau''' \quad \tau' \equiv \tau''}{\tau \rightsquigarrow^{\alpha a} \tau'''}$
---	---

$$\begin{aligned}
\tau_0 &= (C_0, L_0, F_0, M_0, V_0) \\
C_0 &= \lambda t. inc_t(\mathbf{0}) \\
L_0 &= \lambda l. \mathbf{0} \\
F_0 &= \lambda t. \text{EMPTY} \\
M_0 &= \text{COARSE} \\
V_0 &= \lambda t. \mathbf{0} \\
\mathbf{0} &= \lambda t. 0 \in VC
\end{aligned}$$

Figure 4.9: Semantics of traces.

$$\begin{array}{c}
\boxed{\sigma \sim \tau} \\
\text{[STATE EQ]} \\
\mathbb{C} = C \quad \mathbb{L} = L \\
(C, L, F, M, V) \equiv (C, L, (\lambda t. \text{EMPTY}), M', V') \\
\forall i. \mathbb{W}_i = V'_{\gamma(M', i)} \\
\hline
(\mathbb{C}, \mathbb{L}, \mathbb{W}) \sim (C, L, F, M, V)
\end{array}$$

Figure 4.10: Equivalence of basic and optimized states.

operation in the trace, there is no state to which the semantics may step. The theorem below shows that the optimized semantics gets stuck if and only if the original semantics gets stuck.

Theorem 4.1 (Bisimulation).

1. If $\sigma \sim \tau$ and $\sigma \rightarrow^\alpha \sigma'$, then there exists τ' such that $\tau \rightsquigarrow^\alpha \tau'$ and $\sigma' \sim \tau'$.
2. If $\sigma \sim \tau$ and $\tau \rightsquigarrow^\alpha \tau'$ and $\tau' \equiv (C, L, (\lambda t.\text{EMPTY}), M, V)$, then there exists σ' such that $\sigma \rightarrow^\alpha \sigma'$ and $\sigma' \sim \tau'$.

We will prove this theorem by induction, and at each step we will need to know that if one semantics can make progress, so can the other.

Theorem 4.2 (Single-Step Bisimulation).

1. If $\sigma \sim \tau$ and $\sigma \rightarrow^a \sigma'$, then there exist τ' and τ'' such that $\tau \equiv \tau''$, $\tau'' \rightsquigarrow^a \tau'$, and $\sigma' \sim \tau'$.
2. If $\sigma \sim \tau$ and $\tau \rightsquigarrow^a \tau'$ and $\tau' \equiv (C, L, (\lambda t.\text{EMPTY}), M, V)$, then there exists σ' such that $\sigma \rightarrow^a \sigma'$ and $\sigma' \sim \tau'$.

This theorem corresponds to the following pair of diagrams, where we draw the trace growing down. At each step we extend the current state by applying the single-step theorem.

$$\begin{array}{ccc}
 \sigma \sim \tau & & \sigma \sim \tau \\
 \downarrow \quad \Downarrow & & \downarrow \quad \Downarrow \\
 \sigma' \sim \exists \tau' & & \exists \sigma' \sim \tau'
 \end{array}$$

We begin with a few lemmas. First, we show that the algorithmic compatibility relation is really the same as the semantic compatibility relation.

Lemma 4.3. For all M and nonempty fp , $M \vDash fp$ if and only if $M \vdash fp$.

Proof. First suppose $M \vDash fp$. By analyzing each case for M and fp , we'll show that $M \vdash fp$.

- Suppose $M = (\text{BLOCK } d)$. If $d = 1$, then $M \vdash fp$ follows immediately. Otherwise, $d > 1$. Whenever $i < b$, $i \notin \llbracket fp \rrbracket$, and so we must have $\gamma(M, i) \neq \gamma(M, b)$.

Thus b is the smallest index i such that $i \operatorname{div} d = b \operatorname{div} d$. It follows that $b \equiv 0 \pmod{d}$.

If $e \neq N - 1$, then whenever $e < i < N$, $i \notin \llbracket fp \rrbracket$, and so $\gamma(M, i) \neq \gamma(M, e)$. Thus e is the largest integer i such that $e \operatorname{div} d = i \operatorname{div} d$. It follows that $e \equiv -1 \pmod{d}$.

- Now suppose $M = (\text{STRIDE } d)$. If $d = 1$, the $\gamma(M, i) = \gamma(M, j)$ for all i, j , and so $\llbracket fp \rrbracket = 1..N$, so that $b = 0$, $e = N - 1$, and $k = 1$, and so $M \vdash fp$ follows. Otherwise, $d > 1$. If $b \geq d$, then there exists i such that $0 \leq i < b$ and $i \equiv b \pmod{d}$, that is $\gamma(M, i) = \gamma(M, b)$. But $i \notin \llbracket fp \rrbracket$ since $i < b$. Thus we must have $b < d$. Since $e \in \llbracket fp \rrbracket$, it also follows that $e \equiv b \pmod{d}$. Finally, if $e < N - d$, then there exists i such that $N - d \leq i < N$ and $i \equiv b \pmod{d}$. But $i \notin \llbracket fp \rrbracket$ since $i > e$. Thus we must have $N - d \leq e$.

Now suppose $M \vdash fp$. We'll use case analysis again to show that $M \vDash fp$. Let $i, j \in 1..N$ such that $\gamma(M, i) = \gamma(M, j)$ and $i \in \llbracket fp \rrbracket$.

- Suppose $M = (\text{BLOCK } d)$. Then $i \operatorname{div} d = j \operatorname{div} d$ and $b \leq i \leq e$. Note $b \leq j$ since $b \operatorname{div} d \leq j \operatorname{div} d$ and b is the smallest integer i such that $i \operatorname{div} d = b \operatorname{div} d$. If $j \leq i$, we're done, so suppose $i < j$. If $e = N - 1$, then $j \leq e$ trivially. Otherwise $e \equiv -1 \pmod{d}$. Since $j \operatorname{div} d = i \operatorname{div} d \leq e \operatorname{div} d$, it follows that $j \leq e$ since e is the largest integer i such that $i \operatorname{div} d = e \operatorname{div} d$.
- Suppose $M = (\text{STRIDE } d)$. Then $i \pmod{d} \equiv j \pmod{d}$ and $i \pmod{d} \equiv b \pmod{d} \equiv e \pmod{d}$. Since $b < d$ and there is only one non-negative integer i such that $i < d$ and $i \equiv b \pmod{d}$, it follows that $b \leq j$. Similarly, $N - d < e$ implies $j \leq e$. \square

Next, we need to know that the optimized analysis can represent any access pattern. This is the purpose of FINE mode. We apply our knowledge of algorithmic compatibility from the previous lemma.

Lemma 4.4. For all fp , $\text{FINE} \vDash fp$.

Proof. By Lemma 4.3 it suffices to show that $\text{FINE} \vdash fp$. By [BLOCK ALGO COMPAT], this will follow if $b \equiv 0 \pmod{d}$ and $e \equiv -1 \pmod{d}$. But $d = 1$ for FINE mode. The result follows since $n \equiv m \pmod{1}$ for all $n, m \in \mathbb{Z}$. \square

We now show that if an optimized state is equivalent to an optimized state with empty footprints for each thread, then all pending race checks will succeed.

Lemma 4.5. If $(C, L, F, M, V) \equiv (C', L', (\lambda t. \text{EMPTY}), M', V')$, then for all t and $i \in \llbracket F(t) \rrbracket$, $V_{\gamma(M, i)} \sqsubseteq C_t$.

Proof. Follows from the second hypothesis of [EQUIV COMMIT] and the fact that if $M \models fp$ and $i \in \llbracket fp \rrbracket$, then $\gamma(M, i) \in \rho(M, fp)$. \square

We are now ready to prove Theorem 4.2.

Theorem 4.2 (Single-Step Bisimulation).

1. If $\sigma \sim \tau$ and $\sigma \rightarrow^a \sigma'$, then there exist τ' and τ'' such that $\tau \equiv \tau''$, $\tau'' \rightsquigarrow^a \tau'$, and $\sigma' \sim \tau'$.
2. If $\sigma \sim \tau$ and $\tau \rightsquigarrow^a \tau'$ and $\tau' \equiv (C, L, (\lambda t. \text{EMPTY}), M, V)$, then there exists σ' such that $\sigma \rightarrow^a \sigma'$ and $\sigma' \sim \tau'$.

Proof. Write $\sigma = (\mathbb{C}, \mathbb{L}, \mathbb{W})$, $\tau = (C, L, F, M, V)$. Since $\sigma \sim \tau$, we have $\mathbb{C} = C$, $\mathbb{L} = L$, and $\tau \equiv \tau_{\text{pre}}$ where $\tau_{\text{pre}} = (C, L, (\lambda t. \text{EMPTY}), M', V')$ for some M' and V' such that $\mathbb{W}_i = V'_{\gamma(M', i)}$ for all i .

1. To prove the first part of the theorem, we assume $\sigma \sim \tau$ and that $\sigma \rightarrow^a \sigma'$. By analyzing each possible operation a and show the existence of τ'' and τ' such that $\tau \equiv \tau''$, $\tau \rightsquigarrow^a \tau'$, and $\sigma' \sim \tau'$.

- First, suppose $a = \text{acc}(t, i)$. Since $\sigma \rightarrow^a \sigma'$, it follows that $\sigma' = (\mathbb{C}, \mathbb{L}, \mathbb{W}')$ where $\mathbb{W}' = \mathbb{W}[i := \mathbb{W}_i[t := \mathbb{C}_t(t)]]$. We also have $\mathbb{W}_i \sqsubseteq \mathbb{C}_t$. Define

$$\tau' = (C, L, F', M', V')$$

where $F'(t) = \langle i:i:1 \rangle$ and $F'(s) = \text{EMPTY}$ for all $s \neq t$. Also, define $\tau'' = \tau_{\text{pre}}$. Then $\tau \equiv \tau''$ immediately. Since $\text{EMPTY} \oplus i = \langle i:i:1 \rangle$, it follows that $\tau'' \rightsquigarrow^a \tau'$.

We now show $\sigma' \sim \tau'$. Let $\tau_{\text{post}} = (C, L, (\lambda t. \text{EMPTY}), \text{FINE}, \mathbb{W}')$. Then we have $\tau' \equiv \tau_{\text{post}}$. To see this, first note $\tau' \equiv (C, L, F', \text{FINE}, \mathbb{W})$ by [EQUIV MODE]. Then $(C, L, F', \text{FINE}, \mathbb{W}) \equiv \tau_{\text{post}}$ by [EQUIV COMMIT] since by Lemma 4.4, $\text{FINE} \models fp$ for all fp and since $\mathbb{W}_i \sqsubseteq C_t$ by hypothesis. By transitivity of \equiv , it follows that $\tau' \equiv \tau_{\text{post}}$. Thus $\sigma' \sim \tau'$.

- Next, suppose $a = acq(t, l)$. Since $\sigma \rightarrow^a \sigma'$, it follows that $\sigma' = (\mathbb{C}', \mathbb{L}, \mathbb{W})$, where $\mathbb{C}' = \mathbb{C}[t := \mathbb{C}_t \sqcup \mathbb{L}_l]$. Define $\tau' = (C', L, F, M, V)$ where $C' = \mathbb{C}'$, and it follows immediately that $\sigma' \sim \tau'$ and that $\tau \rightsquigarrow^a \tau'$. Let $\tau'' = \tau$. Then by reflexivity of \equiv , $\tau \equiv \tau''$, as desired.
- Finally, suppose $a = rel(t, l)$. Since $\sigma \rightarrow^a \sigma'$, we have $\sigma' = (\mathbb{C}', \mathbb{L}', \mathbb{W})$, where $\mathbb{L}' = \mathbb{L}[l := \mathbb{C}_t]$ and $\mathbb{C}' = \mathbb{C}[t := inc_t(\mathbb{C}_t)]$. Define

$$\tau' = (C', L', F', M', V'),$$

where $C' = \mathbb{C}'$, $L' = \mathbb{L}'$, and $F' = (\lambda t. \text{EMPTY})$. Also define $\tau'' = \tau_{\text{pre}}$. It follows that $\tau \equiv \tau''$ and that $\sigma' \sim \tau'$. Then since $F'(t) = \text{EMPTY}$, we have $\tau'' \rightsquigarrow^a \tau'$.

2. For the second part, we work in the other direction, still assuming $\sigma \sim \tau$. This time, we suppose $\tau \rightsquigarrow^a \tau'$ and show the existence of σ' such that $\sigma \rightarrow^a \sigma'$ and $\sigma' \sim \tau'$. The proof is again a case analysis on a .

- If $a = acc(t, i)$, then since $\tau \rightsquigarrow^a \tau'$, we have $\tau' = (C, L, F', M, V)$ where $F' = F[t := F(t) \oplus i]$. We are also given $\tau' \equiv (C, L, (\lambda t. \text{EMPTY}), M'', V'')$ for some M'' and V'' . Define $\mathbb{W}'_j = V''_{\gamma(M'', j)}$ and $\sigma' = (\mathbb{C}, \mathbb{L}, \mathbb{W}')$. We immediately have $\sigma' \sim \tau'$. By Lemma 4.5 applied to τ' , we have that $\mathbb{W}'_i \sqsubseteq C_t$. Since the set of indices represented by the footprints is only extended by i , it follows that $\mathbb{W}' = \mathbb{W}[i := \mathbb{W}'_i[t := \mathbb{C}_t(t)]]$. Thus $\sigma \rightarrow^a \sigma'$.
- If $a = acq(t, l)$, then since $\tau \rightsquigarrow^a \tau'$, we have $\tau' = (C', L, F, M, V)$. Define $\sigma' = (C', \mathbb{L}, \mathbb{W})$, and it follows that $\sigma \rightarrow^a \sigma'$ and $\sigma' \sim \tau'$.
- If $a = rel(t, l)$, then since $\tau \rightsquigarrow^a \tau'$, we have $\tau' = (C', L', F, M, V)$. Define $\sigma' = (C', L', \mathbb{W})$, and it follows that $\sigma \rightarrow^a \sigma'$ and $\sigma' \sim \tau'$. \square

Since $\sigma_0 \sim \tau_0$, this theorem shows $\sigma_0 \rightarrow^\alpha \sigma$ if and only if $\tau_0 \rightsquigarrow^\alpha \tau$ for some τ such that $\sigma \sim \tau$. Furthermore, since the basic analysis is known to be precise, this shows that our analysis is also precise.

Chapter 5

Implementation and Evaluation

5.1 Implementation

We have developed a prototype implementation of the ideas from Chapters 3 and 4. Our tool, called SHRINKWRAP, is a modification of the implementation of the FASTTRACK dynamic race detector, which is built in ROADRUNNER. We review some of the details from the implementation of FASTTRACK and ROADRUNNER, following the original descriptions [6, 7].

ROADRUNNER is a framework for developing dynamic analyses for multithreaded software that is written entirely in Java and runs on any JVM. ROADRUNNER inserts instrumentation code into the target bytecode program at class-load time. This instrumentation code generates a stream of events for lock acquires and releases, field and array accesses, etc. Back-end tools, such as FASTTRACK, process this event stream as it is generated. The events generated by a thread in the target program are processed by that thread itself. Thus back-end tools are inherently multithreaded.

ROADRUNNER enables back-end tools to attach instrumentation state to each thread, lock object, and data memory location used by the target program. Tool-specific event handlers update the instrumentation state for each operation in the observed trace and report errors when appropriate. The ROADRUNNER framework provides several benefits. By working exclusively at the bytecode level, ROADRUNNER tools can check any Java program regardless of whether source code is available. In addition, ROADRUNNER's component architecture facilitates reliable comparisons

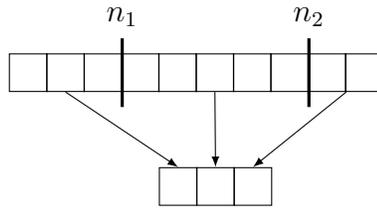


Figure 5.1: Split mode.

between different back-end checking tools, which allows us to evaluate our optimization against the previous version of FASTTRACK.

FASTTRACK is an optimized vector clock-based race detector. We briefly described some of the techniques it uses in Chapter 2. Here, we are concerned only with its treatment of arrays, an issue that is essentially orthogonal to the rest of the algorithm. FASTTRACK uses the standard technique of treating each element of an array separately. Our modification of FASTTRACK implements the compression and redundancy elimination techniques described in the previous chapters to reduce overhead on array-intensive programs.

Our implementation extends the formalism presented above in several ways. First, we support all of Java’s primitive synchronization operations. This extension is essentially handled by ROADRUNNER and FASTTRACK, except that we must ensure that any outstanding indices are committed on each release-like synchronization operation [7]. Second, we distinguish reads from writes, instead of treating all accesses equally. This is necessary to avoid reporting races on simultaneous reads of the same memory location. We also support multiple arrays, as well as accesses to non-array variables, both of which are straightforward.

Footprints are maintained for each thread and array, and footprints are updated on each access. Thus the number of footprint operations is equal to the number of array accesses in the target program, which, in turn, is equal to the number of array race checks handled by the standard implementation of FASTTRACK on the same program.

When the analysis is in fine mode, we do not maintain footprints for efficiency reasons. Instead, we pass each array access to FASTTRACK, hoping to eliminate as much overhead as possible in the case where the target program’s accesses are not well-matched by our modes.

We also added an additional mode, called `SPLIT`, that supports a three-way partition of an array into contiguous blocks, each of arbitrary size. See Figure 5.1. Our motivation was a benchmark that accessed all but the first and last elements of a large array and was subsequently forced into fine mode since those accesses did not match any `BLOCK` or `STRIDE` patterns.

Our implementation also attempts to choose mode transitions that are efficient and compressed. We begin each array in coarse mode, transitioning as necessary to retain precision. Since every array begins in coarse mode, the most common transitions are out of coarse mode. We have not yet explored a general system of transitions among all the modes. Instead, our implementation performs only the transitions that begin in coarse, as well as the block-to-block and stride-to-stride transitions described in Chapter 3.

5.2 Evaluation

We demonstrate the effectiveness of our analysis by evaluating its ability to eliminate redundant checks and precisely compress shadow state. We compare our modified version of `FASTTRACK` with the original in both running time and race check operations, and show that in cases where the arrays are accessed in patterns we can match, our analysis improves performance.

We performed experiments on the following benchmarks: `elevator`, a discrete event simulator for elevators [19]; `hedc`, a tool to access astrophysics data from Web sources [19]; `tsp`, a Traveling Salesman Problem solver [19]; `mtrt`, a multi-threaded ray-tracing program from the SPEC JVM98 benchmark suite [18]; `jbb`, the SPEC JBB2000 business object simulator [18]; `crypt`, `lufact`, `sparse`, `series`, `sor`, `moldyn`, `montecarlo`, and `raytracer` from the Java Grande benchmark suite [10]; the `colt` scientific computing library [2]; the `raja` ray tracer [8]; and `philos`, a dining philosophers simulation [3]. We configured the Java Grande benchmarks to use the number of worker threads reported in Table 5.2 and the largest data set provided. All programs use the same number of threads when executing with and without race detection.

All experiments were performed on a Pogo Linux Atlas server, with dual quad-core AMD Opteron processors and 64GB of memory, running Ubuntu 12.04 and Sun's Java HotSpot 64-bit VM, version 1.6.0. The timing results include class loading time,

Program	Race Checks		Shadow States	
	FASTTRACK (Count)	SHRINKWRAP (\times FASTTRACK)	FASTTRACK (Count)	SHRINKWRAP (\times FASTTRACK)
colt	57,136,376	0.01	424,294	0.2
crypt	1,000,000,351	0.0000004	150,000,123	0.0000006
lufact	7,531,601,524	0.66	4,008,018	1.0
moldyn	6,539,897,621	0.03	167,202	0.63
montecarlo	959,513,963	0.0006	180,026,984	0.003
mtrt	6,076,081	0.15	621,011	1.0
raja	231	1.0	150	1.0
raytracer	1,185,304,894	0.00000007	250,300	0.0003
sparsemm	3,640,498,828	0.42	15,996,750	0.71
series	4,000,064	0.06	2,000,025	0.13
sor	2,423,081,292	0.67	4,002,022	0.75
tsp	269,708,137	0.21	115,605	1.5
elevator	5,664	0.66	502	1.0
philo	165	0.8	10	1.0
hedc	82,396	0.01	44,476	0.01
jbb	397,771,502	0.71	38,300,276	0.26
Geo. Mean		0.02		0.09

Table 5.1: Benchmark shadow race check operations and state allocation.

instrumentation by ROADRUNNER, and execution of the target program.

Shadow state allocation and operations. Table 5.1 contains our data on the operations performed by both the unoptimized and optimized versions of FASTTRACK. We report the total number of accesses handled by FASTTRACK without compression in the first column. The second column shows the proportion of accesses handled by FASTTRACK when compression is enabled. That is, the second column shows how many commit operations SHRINKWRAP performs. The third column shows the total number of shadow states required by FASTTRACK to check the program. This is essentially the number of distinct array elements touched by the target program. The fourth column shows the proportion of shadow states required with compression enabled.

SHRINKWRAP is able to reduce the number of accesses checked by FASTTRACK by at least 90% on 7 out of the 16 benchmarks (*colt*, *crypt*, *moldyn*, *montecarlo*, *raytracer*, *series*, and *hedc*). At least three other programs (*philo*, *raja*, and

Program	Threads (num)	Base Time (sec)	Instrumented Time		
			EMPTY	FAST TRACK	SHRINK WRAP
colt	11	15.9	1.2	1.3	1.1
crypt	7	1.2	12.4	28.7	17.9
lufact	4	5.7	2.0	29.1	12.5
moldyn	4	7.9	7.3	17.1	23.8
montecarlo	4	5.6	3.5	5.7	4.5
mtrt	5	0.4	12.2	13.3	13.3
raja	2	0.4	9.9	11.2	11.2
raytracer	4	5.0	4.3	20.1	23.0
sparsemm	4	4.8	8.2	26.1	26.8
series	4	573.0	1.0	1.0	1.0
sor	4	2.4	2.0	4.9	5.2
tsp	5	0.5	7.5	11.9	13.5
elevator*	5	5.0	1.2	1.3	1.3
philo*	6	8.0	0.3	0.3	0.4
hedc*	6	4.9	1.5	1.5	1.7
jbb*	5	73.0	1.2	1.2	1.2
Geo. Mean			4.4	9.4	8.6

Table 5.2: Benchmark running times. Programs marked with ‘*’ are not compute-bound and are excluded from averages.

`elevator`) are not array-intensive, and so we do not expect any benefit from our technique. The remaining programs may be somewhat array-intensive but their access patterns cannot be exploited by SHRINKWRAP. For example, `sparsemm` is a sparse matrix multiply routine that generates a random access sequence for its arrays at run time. In general, SHRINKWRAP is quite effective at reducing the total number of accesses that need to be checked for races by the underlying detector.

Running Times. Table 5.2 contains the results of our timing experiments. For each program we report the number of threads, and the uninstrumented running time. We also list slowdowns (i.e., multiples of the base time) for three analyses: the EMPTY tool performs no analysis and measures the overhead due to the ROADRUNNER framework; the FASTTRACK standard implementation; and the SHRINKWRAP implementation of our technique. Geometric means of slow-downs are also reported. Each timing benchmark was averaged across 5 runs.

The four programs that are marked with ‘*’ are not compute bound and we do not include them in our averages. These programs include timed delays and other operations that can significantly affect the timing results. For example, some of these programs are actually sped up when instrumented by ROADRUNNER because the instrumentation code changes the behavior of the thread scheduler.

FASTTRACK and SHRINKWRAP differ only in the way they handle array accesses. SHRINKWRAP performs footprint bookkeeping on each array access, while FASTTRACK performs race checks. In those benchmarks where SHRINKWRAP is slower, the bookkeeping costs outweigh the savings of the eliminated race checks. For compute-bound programs that do not use arrays intensively, such as *series*, the performance of SHRINKWRAP is indistinguishable from vanilla FASTTRACK, as one would expect.

It is important to note that a race check for FASTTRACK does not necessarily require a vector clock operation, and thus in some cases the unoptimized operations are quite fast. This plays out in several of the benchmarks, where the number of race check operations and states are reduced significantly, but the running time does not see nearly as large a reduction. This is partially due to the fact that FASTTRACK has already made some of these operations efficient and that building footprints dynamically does induce some overhead. There are also effects due to caching and JIT optimization that are difficult to quantify or account for. In particular, the complexity of some of the analysis in SHRINKWRAP may make the HotSpot compiler reluctant to inline some methods that are inlined in FASTTRACK. This can result in significant slowdowns.

SHRINKWRAP is more time-efficient on 4 out of the 12 benchmarks compute-bound benchmarks (*colt*, *crypt*, *lufact*, and *montecarlo*). On several others, there is no significant difference between FASTTRACK and SHRINKWRAP (e.g., *mtrt*). Finally, there are a few programs for which SHRINKWRAP performs significantly worse than FASTTRACK (*moldyn*). This case is especially surprising given that the results of Table 5.1 showed that SHRINKWRAP eliminated the vast majority of the accesses. We believe SHRINKWRAP can be implemented to give better running time performance but have not explored how to optimize its code further to date. We discuss several benchmark programs in more detail below.

- *crypt* is a cryptographic benchmark that performs an encryption followed by a decryption using several worker threads that divide up an array into blocks.

We mentioned this benchmark briefly in Chapter 2 and presented a high-level description of its algorithm there. FASTTRACK reports that it is already able to eliminate essentially all vector clocks from the analysis of this program [6]. Thus, it is even more impressive that SHRINKWRAP is able to improve the slow-down by around 40%. We attribute this success to the utterly perfect matching of the program’s access pattern to our block mode, especially given that the program has relatively few synchronization points, we are able to absorb a large number of the accesses without further computation.

- `lufact` is a financial simulator. Neither FASTTRACK nor SHRINKWRAP report significant reductions in the number of shadow states allocated, nor does SHRINKWRAP eliminate more operations than FASTTRACK does.
- `raytracer` allocates many relatively small arrays. Thus the extra computation required to record footprints and keep track of compression mode is repeated for each array and cannot be paid for by eliminating race checks because the arrays are so small.
- `sparsemm` performs a sparse matrix multiply. The matrices are randomly generated at runtime, and so the access sequence into the sparse representation is random. This provides a good stress test and sanity check on any array-focused optimizations, since the benchmark contains a large array that is accessed randomly. SHRINKWRAP does not perform significantly worse than FASTTRACK.
- `moldyn` simulates molecular dynamics, using several multi-dimensional arrays to keep track of the forces, positions, etc. This is the one example where SHRINKWRAP performs significantly worse than FASTTRACK, despite the optimization that no footprints are constructed in fine mode. Because the target program uses arrays whose longest dimension is only 2048, the overhead of constructing footprints cannot be offset by savings in race checks.

In general, we find that for SHRINKWRAP to be more time-efficient than FASTTRACK, two conditions must be satisfied. First, the program must use patterned array accesses that can be exploited by the analysis. Second, commits must operate on large enough footprints to offset the cost of building them at runtime. That is, the amortized cost of building and committing footprints must be lower than the amortized cost of a race check in FASTTRACK.

Further Performance Improvements. We briefly outline several approaches to further improving the performance of SHRINKWRAP.

- Every array access in the target program must attempt to extend the current footprint with the new index. To date, we have not attempted to optimize our implementation of footprint extension, but we expect that such an effort would be worthwhile.
- SHRINKWRAP synchronizes all threads accessing an array when that array undergoes a mode transition. This synchronization overhead can be significant if the array is being accessed heavily. A more sophisticated synchronization discipline may be able to reduce this overhead.
- Instrumentation influences the way the HotSpot compiler optimizes the byte-code of the target program. By examining how SHRINKWRAP affects the compiler and then tuning our implementation in response, it should be possible to mitigate some of the initial slowdown.

Chapter 6

Conclusions

6.1 Contributions

We have developed an analysis that detects and eliminates two types of redundancy commonly found in dynamic race detectors when run on array-intensive programs. Namely, that there are many repeating shadow states in the shadow array, and that many race checks are redundant. Our analysis compresses the shadow array using one of several modes; it also delays race check operations until the next synchronization operation in order to take advantage of this compression. We have implemented our analysis in a state-of-the-art race detector and shown that we can improve performance when the target program accesses arrays in a pattern we recognize. SHRINKWRAP is able to eliminate the vast majority of the accesses that must be checked by the underlying race detector on almost half of our benchmark programs. In other cases, it is either not able to recognize a pattern in the program's accesses, or the program is not array-intensive. However, even when SHRINKWRAP can significantly reduce the number of accesses checked by the underlying algorithm, it is not always more time-efficient due to the overhead of constructing footprints. We believe that SHRINKWRAP could be made more efficient by tuning its implementation in response to the HotSpot compiler.

6.2 Future Work

The overhead of constructing footprints could be completely eliminated by inferring access patterns statically. These patterns could then be reported directly to the run time system, which could then eliminate the dynamic analysis required to discover them. Together with the fact that we delay race checks until the next synchronization operation, this has the potential to move essentially all analysis code out of the inner loops of benchmarks, which could yield large performance improvements. The key part of such a system would be the design of an effective static analysis for precise access pattern inference. Such a problem is easier than a general static analysis for race freedom because it need only reason locally about a particular loop, rather than considering the program as a whole.

SHRINKWRAP currently backs off into fine mode indefinitely. In other words, as soon as an array has been accessed in a way that SHRINKWRAP cannot match with any of its modes, the array will remain uncompressed for the remainder of the program. In some cases, however, it may be possible to recompress the array at a later time when the accesses become more patterned. There are two problems to be overcome in this direction. First, it is not clear what modes to propose for recompression. Second, checking that an array can be compressed in a given mode is relatively expensive, since it involves a linear scan of the shadow array. Thus only very few proposals can be reasonably considered. It may be useful to continue to build footprints for this purpose, basing the proposed compression mode on the current access pattern. However, the cost of building footprints when there is no pattern to the accesses is quite high, so this approach would require care.

Finally, it may be possible to offload some of the overhead of creating and managing footprints to other cores in the machine. In this case, the target program would be free to continue executing while the race detector worked in the background. This is similar in spirit to garbage collectors that work without interrupting the running program.

Bibliography

- [1] ABADI, M., FLANAGAN, C., AND FREUND, S. N. Types for safe locking: Static race detection for Java. *ACM Trans. Program. Lang. Syst.* 28, 2 (2006).
- [2] CERN. Colt 1.2.0. <http://dsd.lbl.gov/~hoschek/colt/>.
- [3] ELMAS, T., QADEER, S., AND TASIRAN, S. Goldilocks: A race and transaction-aware Java runtime. pp. 245–255.
- [4] FLANAGAN, C., AND FREUND, S. N. Detecting race conditions in large programs. In *PASTE* (2001), pp. 90–96.
- [5] FLANAGAN, C., AND FREUND, S. N. Type inference against races. In *SAS* (2004), pp. 116–132.
- [6] FLANAGAN, C., AND FREUND, S. N. Fasttrack: efficient and precise dynamic race detection. In *PLDI* (2009), pp. 121–133.
- [7] FLANAGAN, C., AND FREUND, S. N. FastTrack: Efficient and precise dynamic race detection. *Commun. ACM* 53, 11 (2010).
- [8] FLEURY, E., AND SUTRE, G. Raja, version 0.4.0-pre4. Available at <http://raja.sourceforge.net/>, 2007.
- [9] ITZKOVITZ, A., SCHUSTER, A., AND ZEEV-BEN-MORDEHAI, O. Toward integration of data race detection in DSM systems. *J. Parallel Distrib. Comput.* 59, 2 (1999), 180–203.
- [10] JAVA GRANDE FORUM. Java Grande benchmark suite. Available at <http://www.javagrande.org>, 2008.

- [11] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (1978).
- [12] MATTERN, F. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms* (1988).
- [13] NAIK, M., AND AIKEN, A. Conditional must not aliasing for static race detection. In *POPL* (2007), pp. 327–338.
- [14] NAIK, M., AIKEN, A., AND WHALEY, J. Effective static race detection for Java. In *PLDI* (2006), pp. 308–319.
- [15] O’CALLAHAN, R., AND CHOI, J.-D. Hybrid dynamic data race detection. In *PPOPP* (2003), pp. 167–178.
- [16] POZNIANSKY, E., AND SCHUSTER, A. Efficient on-the-fly data race detection in multithreaded C++ programs. In *PPOPP* (2003), pp. 179–190.
- [17] SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. E. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.* 15, 4 (1997), 391–411.
- [18] STANDARD PERFORMANCE EVALUATION CORPORATION. SPEC benchmarks. <http://www.spec.org/>, 2003.
- [19] VON PRAUN, C., AND GROSS, T. Static conflict analysis for multi-threaded object-oriented programs. pp. 115–128.
- [20] YU, Y., RODEHEFFER, T., AND CHEN, W. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP* (2005), pp. 221–234.