2017

Sound Thread Local Analysis for Lockset-Based Dynamic Data Race Detection

Samuila Mincheva
sminchev@wellesley.edu

Sound Thread Local Analysis for Lockset-Based Dynamic Data Race Detection

Samuila Mincheva

Submitted in Partial Fulfillment
of the
Prerequisite for Honors
in the Computer Science Department
under the advisement of Benjamin P. Wood

May 2017

Abstract

Multithreading is a powerful model of parallel and concurrent programming. However, the presence of shared data leaves multithreaded programs vulnerable to concurrency errors such as data races, where two threads access and modify the same data concurrently and without synchronization. Data races lead to unpredictable program behavior and can be a source of data corruption. This work improves the precision of lockset-based dynamic data race detection without compromising soundness. Typically, lockset-based algorithms are sound but extremely imprecise. The algorithms presented in this work improve the precision of such algorithms by including thread-tracking information. Thread tracking helps detect patterns of intermittent thread-locality of shared data and eliminate false errors while still reporting all true errors. Experimental results show that thread-local analysis preserves soundness and improves precision of lockset-based data race detection by an average of 82%, with run-time slowdowns of less than 17%.

Acknowledgements

I would like to thank Ben Wood for his continuous guidance, not only throughout this project, but throughout my entire academic career at Wellesley. I would also like to thank my committee members, Ashley DeFlumere, Alex Diesl, and Lyn Turbak, for their invaluable insight and comments. I am immensely grateful to my major advisor, Sohie Lee, for introducing me to the world of Computer Science in the most inspiring and exciting of ways, and for always finding the time to give me advice, encouragement, and chocolate. Finally, I am grateful to everyone else in the Computer Science department who has been there for me over the past four years, and to all my friends and family who supported me along the way.

Contents

Chapter 1

Introduction

This work presents a series of Thread-Aware Lockset algorithms that improve the precision of lockset-based dynamic data race detection while maintaining soundness. Data race detection is crucial in parallel and multithreaded computing, where the presence of shared data leaves programs vulnerable to a variety concurrency errors.

Concurrent and parallel programming lie at the core of modern computing, since they make possible the efficient processing of large amounts of data simultaneously. A common model of parallel computing is *multithreading*. Single-threaded program execution follows a sequential order in which each instruction is executed only after the completion of the instruction that immediately precedes it. In contrast, the multithreaded programming model allows machines to execute different threads simultaneously by interleaving the sequences of code from each thread throughout the program execution.

In the multithreaded model, each thread has its own private registers and stack, but all threads share the same heap. This means that it is possible for more than one thread to have access to the same memory location. Thus, performance is improved by allowing tasks over the same shared data to be split between different threads. However, it can cause problems if the same memory location is accessed and modified by different threads at the same time, and can lead to incorrect, and unpredictable, program behavior and concurrency errors such as data races. Since different program executions might lead to different interleavings of the threads, reproducing the exact interleaving that leads to erroneous behavior is often time-consuming, as there is no way to guarantee what order threads will execute their code.

Therefore, debugging multithreaded programs cannot be accomplished in the usual style of single-thread debugging, which relies heavily on reproducibility. Special tools need to be developed to allow for the efficient debugging of multithreaded software.

1.1 Problem

1.1.1 Performance, Soundness and Precision

When evaluating data race detection algorithms, the most common considerations are concerned with the tool's performance, precision, and soundness.

Precision refers to the absence of false alarms; a *precise* tool would report no false alarms, meaning that any condition flagged as a data race is actually a data race. Soundness refers to the tool's ability to detect all data races; a *sound* tool is one that guarantees to report all data races in a program. While in theory it would be ideal to have a data race detection tool that is both precise and sound, the checks required to ensure both properties significantly can slow down program performance in practice.

1.1.2 Problem Statement

Due to the ubiquitous presence of multithreaded programming, there exist a variety of tools to detect and report data races. However, such tools have a number of limitations. One of the most common types of dynamic data race detection involves the computation of guarding locksets for each memory location. This approach relies on verifying that all accesses to a location are consistently guarded by at least one mutual exclusion lock. Lockset-based tools are sound, but very imprecise, since they fail to detect any other type of synchronization mechanisms in the program. Other tools that achieve perfect precision and soundness involve complicated computations and tracking large amounts of data.

In practice, a lot of data in multithreaded programs is thread-local, which means it is only accessed by one thread. Thread-local data does not need to be protected by any locks, since memory accesses by the same thread are free of data races by definition. Lockset-based data race detection tools report false alarms on such data, since they lack the information to determine if data is shared or not. While there exist thread-local filtering tools, they have a number of shortcomings. Since they are in the form of prefix filters, which means that they can detect thread locality only at the beginning of a datum's life, they fail to detect patterns of extensive intermittent thread-local reprivatization. Crucially, the current implementation of these filters can compromise the soundness of the tools they are used with, which leads to missed data races.

1.2 Goals

The goal of this work is to develop new thread-aware lockset-based algorithms and compare their precision and performance to those of existing lockset tools in order to characterize the improvement

offered by thread analysis. I aim to eliminate the risk of unsoundness posed by current thread-local filters by incorporating the thread analysis as part of the lockset algorithms. The second goal of the thread-aware lockset-based algorithms is to support the classification of data as thread-local at any point during the data's life, not just at the beginning.

The long-term goal of this work is to gain insight into how lockset-based data race detection can be made more effective. It is not within the scope of this project to develop a lockset-based algorithm that provides significant improvements over existing fast and precise algorithms. My goal is rather to explore how thread-local analysis can be used to enhance lockset-based algorithms. Combined with other possible optimization that could eliminate some of the performance overhead of maintaining both a thread and a lockset for each memory location, thread-local analysis might make such significant improvements possible in the future.

1.3 Contributions

This work presents a series of novel algorithms that use thread analysis to provide a more robust model of lockset-based data race detection. The algorithms are able to capture patterns of intermittent or eventual thread-locality that simple lockset algorithms cannot detect, without the need for the complex computations required by vector-clock-based data race detection.

The work develops a series of tools based on the algorithms and evaluates their precision, soundness and performance on a suite of multithreaded Java benchmarks. Experimental evaluation shows that sound thread analysis improves the precision of lockset-based data race detection relative to the canonical algorithm by 82% on average, while incurring an average slowdown of only 17%. Case studies of two benchmarks characterize the patterns of execution for which lockset-based tools report false alarms and the ways in which thread analysis can reduce this imprecision.

Overall, these results demonstrate that sound thread-local analysis opens up new potential for effective lockset-based data race detection.

1.4 Outline

The work is presented as follows:

Chapter 2 covers the background and motivation for this project, as well as existing approaches and theory to data race detection.

Chapter 3 presents the thread-aware data race detection algorithms developed as part of this work, and provides an argument for their soundness.

Chapter 4 outlines the implementation of the algorithms for multithreaded Java programs.

Chapter 5 evaluates the precision and performance of the tools developed for each algorithm.

Chapter 6 describes areas for future work.

Chapter 7 concludes.

Chapter 2

Background and Motivation

2.1 Motivating Example

To illustrate the importance of data race detection, consider the simplified example of an online voting system. Different voters can cast their vote at the same time in different threads. If they cast a vote for the same candidate and the voting system does not handle the concurrency correctly, it is possible that the way in which the different voting threads access and update the vote count for that candidate leads to a wrong final result. The thread interleaving demonstrated in Figure 2.1 would produce such an error.

Thread 1	Thread 2	Value of `O.count`
`x = O.count  // x = 45`		45
	`y = O.count  // y = 45`	45
	`y += 1       // y = 46`	45
	`O.count = y  // y = 46`	46
`x += 1       // x = 46`		46
`O.count = x  // x = 46`		46

FIGURE 2.1: Multithreaded program execution containing an example of a concurrency error. (Time flows down.)

Both threads follow a simple model of updating the global `O.count` variable: they store the global count into a local variable, `x` or `y`, increment the local variable by `1`, and then store the updated result back in `O.count`. However, an error occurs when execution switches to `Thread 2` before `Thread 1` has finished performing its update, which leads to an incorrect global count being stored at the end.

This error occurs because the update procedure that the threads follow lacks *atomicity*. A block of code is atomic if its execution is not affected by and does not interfere with concurrency. This means that regardless of whether there are other threads accessing the same memory location, the behavior of the atomic section of code is deterministic, and any shared data that is read or modified by the critical section of code is not modified by another thread during the execution of the critical section [1, 2]. In this example, the three lines of code that constitute the procedure to update `O.count` should be executed atomically, because they represent one logical operation as a whole. However, there is no mechanism employed to guarantee the atomicity of the operations, which leaves the code vulnerable to *data races* [3]. Data races are pairs of concurrent accesses to the same memory location that execute in a nondeterministic order and can lead to unpredictable results. In this case, both `Thread 1` and `Thread 2` read and modify `O.count` concurrently, which creates a data race condition that violates the atomicity of the code. Therefore, the program execution can contain concurrency errors that result in an erroneous final result in `O.count`.

It is easy to imagine how a generalization of this problem to a larger scale system can have dire consequences.

2.2 Happens-Before and Data Races

Data races are defined in terms of the happens-before relationship, which was first developed by Lamport in [4]. The *happens-before* relation, denoted as $\prec$, provides a partial ordering of events in a distributed system, where $A \prec B$ if one of the following holds:

1. A and B are events in the same thread and A comes before B sequentially,
2. A is the release of a lock from one thread and B is the acquire of the same lock by a different thread,
3. There is a third event C such that $A \prec C$, and $C \prec B$.

Other synchronization mechanisms, such as fork-join, can be used to established a happens-before relationship between events as well.

In this model, two events are *concurrent* if neither $A \prec B$ nor $B \prec A$ holds. A data race can then be defined using this notion of concurrency. A *data race* occurs in a multithreaded program when all of the following occur:

1. Two or more threads access the same shared memory location,
2. At least one of the accesses is a write,
3. The accesses happen concurrently.

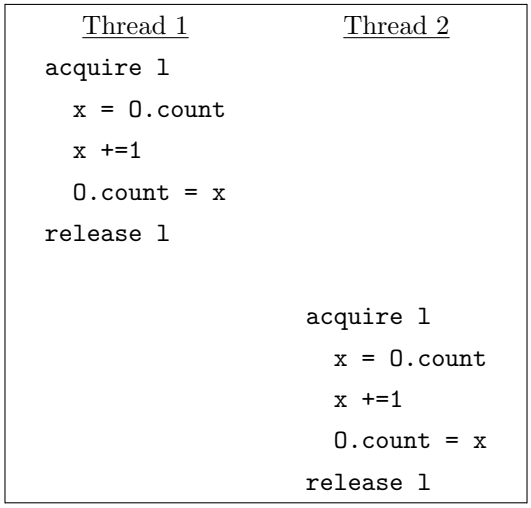

FIGURE 2.2: Example of a happens-before edge introduced by the use of locks.

To illustrate the practical application of this definition more clearly, consider the example from the previous section. The program execution in Figure 2.1 is an example of a data race, because both threads write to the `O.count` variable, and there is no happens-before edge established between the accesses. A happens-before edge could be introduced by the use of a lock guarding `O.count`, as shown in Figure 2.2. In this example, both `Thread 1` and `Thread 2` have to acquire the lock `l` before they can access `O.count`. The actions of acquiring and consequently releasing the lock ensure the atomicity of the increment operation and introduce a happens-before relationship between the two accesses, which guarantees they will be free of data races.

2.3 Data Race Detection Tools

2.3.1 Static Analysis

Static data race detection algorithms analyse code before it is run and try to determine memory accesses that can cause a data race. An example of such an algorithm is the one used in rccjava [5]. It extends the type system to capture common synchronization patterns and detect data races by tracking the locks guarding each memory access. Although static data race detection tools trivially incur no run-time overhead for the program, they are often conservative in their error reporting. This means that, in order to be sound, they need to be imprecise since they lack the insights into program behavior that dynamic tools have access to. Therefore, a large number of the errors reported by static data race detection tools are typically false positives, which causes programmers to use up time and resources in identifying those false positives, or finding ways to work around the tool's reports.

An exception to this model is the algorithm presented in [6], which is a static race detection algorithm that compromises soundness in order to guarantee near-perfect precision. This is achieved by employing a combination of successive static analyses that effectively reduce the number of memory access pairs that could be involved in a data race. The four consecutive steps compute the *reachable pairs*, the *aliasing pairs*, the *escaping pairs*, and the *unlocked pairs* of memory accesses in a program. Each step further refines the results calculated by the previous one. The reachable pairs computation filters memory accesses based on the fact that only accesses reachable from a thread that is itself reachable from the *main* can be involved in a data race. The second filter, aliasing pairs, employs the fact that a pair of memory accesses can be involved in a data race only if they access the same location. Next, the escaping pairs computation filters out accesses to thread-local data; that is, data that is only accessed by one thread. Finally, the unlocked pairs computation filters out pairs of accesses that are safe from data races based on holding a common set of locks. Through this careful analysis, the algorithm filters out most of the false alarms that would otherwise be raised by a static race detector. However, it does so at the expense of soundness, as some of the assumptions necessary for this type of static analysis rely on estimations rather than provable facts about the program.

2.3.2 Vector Clock Algorithms

A large subset of dynamic data race detection algorithms relies on the notion of happens before as defined by Lamport [4] to perform run-time data race checks. In practice, however, the logical clocks proposed by Lamport are not enough to precisely determine if a data race occurs. Mattern expands the happens-before model by introducing the notion of Vector Clocks [7]. A *vector clock* is a vector of recordings of relative time for each thread in a program. Vector clocks are a way for each thread in a system to track the relative time of each other thread in the system, which makes it possible to determine if there is ever a data race. Algorithms that follow that model perform checks to establish whether conflicting memory access are separated by synchronization events.

The algorithm implemented in the FastTrack tool implements the idea of vector clock enforced detection [8]. Each thread has a vector clock maintaining the current state of the entire system, or rather the thread's knowledge of the state of the system; that is, each thread has a map from thread to current time for each other thread in the program. A thread updates its own entry in its vector clock when it performs a synchronization event such as a releasing a lock. The synchronization objects, or locks, are also augmented with a vector clock, which is updated accordingly when events using that lock occur. This allows threads to communicate and update their vector clocks' entries for other threads representing the current time of program execution.

Vector clock algorithms are both sound and precise for the current execution of the program. This means that they report all data races that occur in this execution, with no false alarms, but do not

report all data races that could occur in all feasible executions of the program. However, the runtime overhead incurred by performing the checks necessary to establish the happens-before relations between different memory accesses in the program often mean that such tools slow down program execution significantly; FastTrack cites an average slowdown of 8.5 times [8].

2.3.3 Lockset Based Algorithms

Lockset based data race detection algorithms are founded on the common locking model in which each memory location is consistently guarded by at least one lock. A thread must acquire the lock for a memory location before it accesses that memory location, then release it after it has performed the operations that represent one logical atomic access, as judged by the programmer. The lock ensures that no other thread can intervene and access or modify the memory location while the original access thread still holds the lock, which prevents data races by definition: no two threads can have concurrent access to a memory location, as long as it is consistently guarded by the lock.

Eraser is the canonical dynamic data race detector that uses a lockset-based approach [9]. The basis of the algorithm is quite simple: it enforces a locking discipline in which each memory location must be consistently guarded by at least one lock throughout the entire program. This means that there must be a non-empty set of locks, or *lockset*, that is held consistently at each access to that memory location. A data race is reported when it becomes clear that there is no such lockset for a memory location. Algorithm 2.1 presents the basic Eraser algorithm for a memory access to a variable x by a thread t, as described in [9].

Algorithm 2.1 Eraser Lockset

```
1: function ACCESS(x, t)
2:     LS(x) := LS(x) ∩ LS(t)
3:     if LS(x) = ∅ then
4:         issue a warning
5:     end if
6: end function
```

Eraser's algorithm is implemented by tracking a lockset shadow variable for each memory location. At each access to that location, the intersection of the stored lockset and the lockset held by the current thread is calculated. If the intersection is not empty, this means that there has been at least one lock consistently guarding the location so far, and the lockset shadow variable is updated to that new intersection. If, on the other hand, the intersection of the recorded and current lockset is empty, the program signals a data race for that memory access, as there is no identifiable guarding lock for the location.

Figure 2.3 demonstrates the way in which the Eraser algorithm detects data races. When `Thread 1` first accesses the memory location x, x's guarding lockset is updated to be the set of locks currently held by `Thread 1`, i.e. $\{m_1, m_2\}$. Then, on the subsequent access by `Thread 2`, x's lockset is

Thread 1	Thread 2	LS Guard State
`acquire` m_1		
`acquire` m_2		
`write to` x		$\{m_1, m_2\}$
`release` m_2		
`release` m_1		
	`acquire` m_2	
	`acquire` m_3	
	`write to` x	$\{m_2\}$
	`release` m_3	
	`release` m_2	
`acquire` m_1		
`write to` x		$\{\} \Rightarrow$ Error
`release` m_1		

FIGURE 2.3: Example of the way Eraser detects data races.

updated to be $\{m_2\}$, which is the intersection of the two locksets: the one previously recorded for that location, and the one that is held by the current access thread. Then, when `Thread 1` accesses x again, the same procedure is performed, and the intersection between the recorded and current access lockset is taken. However, this results in an empty lockset. This signifies to Eraser that a potential data race can occur at that access, so it reports an error.

Thread 1	Thread 2	LS Guard State
`acquire` m_1		
`acquire` m_2		
`write to` x		$\{m_1, m_2\}$
`release` m_2		
`release` m_1		
	`acquire` m_2	
	`acquire` m_3	
	`write to` x	$\{m_2\}$
	`release` m_3	
	`release` m_2	
`acquire` m_1		
`acquire` m_3		
`write to` x		$\{\} \Rightarrow$ Error
`release` m_3		
`release` m_1		

FIGURE 2.4: Example of a situation in which Eraser reports a false positive.

Figure 2.4 shows an example of a series of accesses that are also data race free, but for which Eraser reports an error. It behaves in the exact same way as in the example in Figure 2.3, even though this example is not a data race condition, as there is always at least one common lock between each pair of accesses from different threads. However, because it always records the intersection of the current access lockset and the recorded lockset, Eraser does not recognize this as a valid and safe locking discipline and instead reports a data race for the last access.

The base Eraser algorithm is sound, as it guarantees to report any data races that could occur in a program. However, as demonstrated above, it is not precise because it only tracks the locking behavior of a program, and even then it fails to detect some locking patterns that would still guarantee data race free execution. In practice, there are many other commonly used synchronization mechanisms, such as fork-join. Those could be used to write a program free of data races without the use of locks. This would lead Eraser to report many false positive errors because of its sole reliance on locksets.

2.3.4 Hybrid Algorithms

Various data race detection tools rely on a combination of multiple approaches to provide better performance, soundness and precision. Many dynamic tools utilize static analysis to filter out unnecessary run-time checks, which can lead to improved performance. Furthermore, there are some hybrid tools that combine the lockset and happens-before based approaches, which allows them to be both sound and precise, even though that usually comes at the cost of decreased performance.

For example, the Goldilocks algorithm is a dynamic data race detection algorithms that uses locksets to establish happens-before relations in a program execution [10]. Though lockset-based, the Goldilocks algorithm handles other synchronization disciplines such as software transactions. It also stores other metadata such as the last thread to access a memory location, which improves the precision of its analysis. This makes it unlike most other lockset based algorithms which are imprecise, but also allows it to maintain performance better than that of traditional happens-before algorithms that use vector clocks in their analysis. Furthermore, the algorithm uses static analysis of the code to improve run-time performance.

Acculock is another example of a happens-before and lockset hybrid algorithm [11]. It uses epoch-based data race detection similar to FastTrack to establish weak happens-before edges in a program execution, ignoring the locking discipline of the program. If a violation is discovered in the establish relations, it uses a lockset algorithm to filter out those accesses that are actually protected by the locking discipline. This way it guarantees soundness while improving over the precision of traditional lockset-based algorithm. It maintains performance comparable to FastTrack, but still slower than Eraser's when implemented with the same framework.

2.4 Optimizations to Data Race Detection Algorithms

2.4.1 Escape Analysis

There are some observations about program behavior that can help improve the precision and performance of dynamic data race detection tools. A key observation is that all data is initialized as local to a single thread; i.e., it is initially private, or *thread-local*, data. Data escapes when it becomes available to more than one thread. Therefore, data race detection algorithms can be optimized to perform checks only on shared data — that is, only data that has escaped.

2.4.1.1 Eraser

Eraser uses this knowledge to filter out unnecessary and expensive lockset checks on data that is guaranteed to be race-free. Data can exist in four states. All data is initialized in the *Virgin* state, and transitions to the *Exclusive* state once it has been written to by a thread. Any subsequent accesses by the same thread maintain it in the *Exclusive* state, because by definition no data race could occur between them. No lockset information is recorded during the *Virgin* and *Exclusive* states. A read from a different thread changes the data's state to *Shared*, in which lockset recording begins. As long as no thread writes to the location, the same state is maintained and no data races are reported even if the lockset becomes empty, because the data has not been modified in a situation in which a data race could occur. The last state is the *Shared-Modified* state. The transition to that state happens upon a write from a new thread if in the *Exclusive* state or a write from any thread in the *Shared* state. The checks performed in the *Shared-Modified* state are those in the base Eraser algorithm; the intersection of the current and the recorded lockset is taken, and data races are reported if it ever becomes empty.

This enhanced model allows Eraser to filter out many unnecessary checks, which improves not only performance but precision as well. However, it does leave it vulnerable to the case in which the first shared-modified access to a new memory location is unsafe, because no previous lockset information would have been recorded for it. This is demonstrated in Figure 2.5. Therefore, even though the filter improves performance and precision, it impacts the soundness of the subsequent analysis.

2.4.1.2 TRaDe

Another algorithm that deals with escape analysis is TRaDe [12], a topological race detector that uses vector clocks. Data in the TRaDe algorithm is monitored using the underlying garbage collection to determine whether it is reachable by more than one thread at any point in the execution. Data is transitioned from private to shared whenever a reference to a previously local memory location is posted to a global location, even if it has only been directly accessed by one thread. This is

Thread 1	Thread 2	Guard Tool for `x.m`	Guard State for `x.m`
`begin`			
`start Thread 2`			
	`begin`		
`x = new X()`		TL	
`x.m = 5`		TL	`Thread 1`
	`acquire l`	TL	`Thread 1`
	`x.m = 13`	TL ⇒ Lockset	{`l`}

FIGURE 2.5: An example trace for which use of the Thread-Local prefix filter would result in unsound analysis. The *Guard Tool* column shows the tool currently guarding the memory location (i.e., TL or Lockset), and the *Guard State* column shows the recorded metadata for the location. Since the first write to `x.m` is while the field is still thread-local, no lockset information is recorded for the field. Therefore, on the first access to `x.m` by Thread 2, the guarding lockset becomes {`l`}, even though the field was previously unprotected and the two accesses to it constitute a data race. Therefore, it is not possible to know whether it forms a data race with any previous access.

different from Eraser's algorithm, which marks data as shared only after it is accessed by a different thread. This distinction of data being accessed versus accessible by more than one thread is crucial in maintaining the soundness of the algorithm.

Thread 1	Thread 2	Status of `x`
`x = new X()`	...	Local to Thread 1
...	...	
`O.f = new X()`	...	*Accessible* to T2 through global object O
	...	
	`O.f.doSomething()`	*Accessed* by T2

FIGURE 2.6: Difference between data being accessed and accessible to more than one thread.

Figure 2.6 illustrates the difference in code for data becoming accessed versus it becoming accessible. When the reference to the locally initialized object `x` is first posted to the global object `O`, the TRaDe algorithm would mark it as shared and begin data race monitoring from that point on, because it is now globally accessible. Eraser, however, would not mark `x` as shared until later in the code, when `Thread 2` first accesses `x` through the reference stored in `O.f`. TRaDe's approach maintains the soundness of the algorithm because it is guaranteed that by the first global access to `x`, it will have already recorded the necessary information to perform a data race check.

2.4.1.3 Other Algorithms

Many other data race detection algorithms, regardless of whether they are based on locksets and happens before, use similar approaches to filter out unnecessary check or reduce runtime overhead. For example, the FastTrack algorithm [8] performs analysis to determine if data is thread local or read shared. It uses the observation that thread-local accesses to data are necessarily ordered, and reads to read-shared data are free of data races if they are ordered with all previous reads. This allows the algorithm to perform constant time and space overhead checks for those cases, as opposed to the usual analysis, which is linear in time and space. It accomplishes this by switching from tracking vector clock metadata to tracking *epochs*, which is a pair of clock and thread identifier of the last access to the data. This significantly improves the efficiency of the algorithm without compromising soundness and precision.

An example of thread local analysis is implemented as part of the RoadRunner framework for dynamic analysis [13]. It facilitates program analysis by allowing memory locations to be extended with shadow data that can be used to perform various checks and analyses. In the case of the thread local extension, the recorded shadow data is the thread identifier of the previous access thread. At each subsequent access to the memory location, the identifier of the thread is checked against the recorded thread identifier; if they are the same, the data is determined to be thread local. If they are different, the data leaves the thread-local state and is passed on to the next stage of the analysis. This is a simple and efficient way to filter out unnecessary and expensive checks on a large amount of data.

2.4.2 Reprivatization

While escape analysis offers optimization opportunities at the beginning of program execution, reprivatization can detect when previously shared data becomes private again, allowing for further performance gain. Much like data begins in a thread-local state and then transition to being thread-shared, it is possible for shared data to change back to a thread-local state at some point during program execution.

Figure 2.7 extends the example from the previous section to illustrate the way in which data can be reprivatized. Here, it is also possible to make the distinction between data being accessed by more than one thread, versus being accessible to more than one thread. Assuming that x is shared between `Thread 1` and `Thread 2`, after `Thread 2` accesses `x` for the last time, that memory location transitions back to being accessed solely by `Thread 1`. However, at that point it is still accessible to `Thread 2`; that changes only after the thread nullifies its last reference to the memory location. At that point, the data becomes accessible only to `Thread 1`.

Similarly to initially thread-local data, reprivatized data is safe to access without synchronization because only one thread is performing accesses to the location. In this example, after `Thread 2`

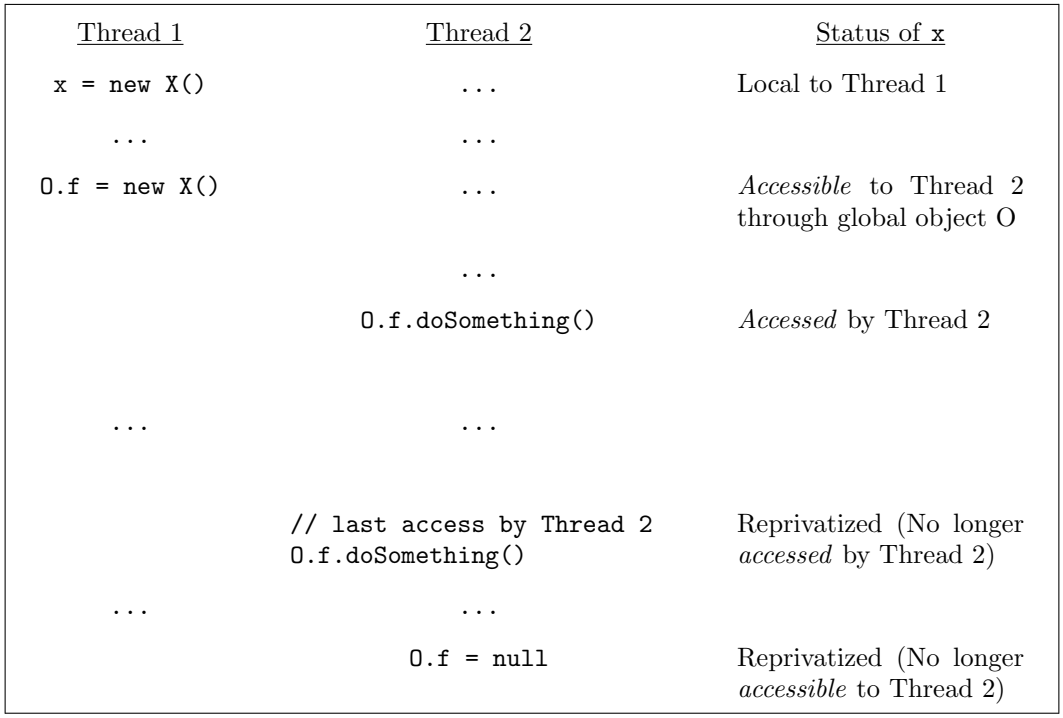

FIGURE 2.7: Data being reprivatized, following the accessed vs. accessible principle.

stops accessing `x`, no data race can occur involving `x`, because `Thread 1` is the only thread that accesses that memory location. By definition, such accesses are guaranteed to be free of data races.

The TRaDe algorithm discussed in the previous section supports reprivatization analysis. Because it periodically tracks how many threads have access to each memory location, it can establish if a location ever goes back to being owned by a single thread. This means that data that used to be shared can become private at a point in program execution when only one thread has access to it. This is an improvement over the prefix optimizations of Eraser, as it provides more opportunity for eliminating unnecessary checks.

Two other algorithms use the Eraser state machine as a basis for their filtering, while extending it with additional transitions and states that allow for reprivatization of data.

The MulticoreSDK data race detector described in [14] employs a state machine based on the state machine used by Eraser. They add two additional transitions which allow data to go back to the *Exclusive* state if it is determined that it is only accessed by one alive thread. Therefore, while less complex than TRaDe's analysis, it still provides performance and precision improvements over the basic algorithms.

Similarly, Pozniansky and Schuster present another state machine that extends Eraser's [15]. They add additional states to the basic state machine presented in Eraser, the most significant of which is the *Clean* state to which data transitions after the presence of a barrier. The employment of this

additional synchronization analysis allows for improving the precision of the algorithm, and serves as a simplified version of reprivatization analysis.

2.4.3 Other Optimizations

In practice, because the problem of detecting data races is computationally hard, there is no single best approach. Most tools base their analysis on one approach and then add optimizations to improve on the quality of their performance, soundness, and precision.

The filtering techniques for the Intel Thread Checker Race Detector described in [16] provide further examples of optimizations to the basic data race detection algorithms. The proposed filters use an Eraser-like state machine to eliminate checks on thread local and read shared data in the vector clock based algorithm the Thread Checker uses. However, they modify the basic Eraser state machine to perform data race checks on the references that trigger transitions from one state to another, which resolves the soundness issue introduced by Eraser's optimization. The algorithm also filters out stack references by a simple memory address check, since stack data is always private. Additionally, they implement a filter that removes duplicate references from the checked conditions. This improves performance, while it guarantees to only remove duplicate data races, meaning it maintains soundness.

2.5 Design Trade-Offs

Existing approaches to data race detection often need to make a compromise between the soundness and the precision of their reports. Theoretically, it seems more intuitive to focus on implementing a sound algorithm, in order to guarantee that all data races will be reported. However, it is important to consider the practical application of the tools: if an algorithm is completely sound, but imprecise to the point where most of the data races it reports are false alarms, it would not be a feasible solution for a debugging tool, as it would mean that a lot of the time and effort spent on analyzing its results are wasted on the false alarms. Thus, in practice in some cases it may be acceptable, and even desirable, to allow small compromises in soundness in order to improve precision significantly.

Different algorithms make different guarantees for the soundness and precision of their tools. For example, there are tools that are precise with regard to the current execution of the program. This means that they report data races only if they actually occurred in this particular execution, as opposed to data races that could occur in other executions with different interleavings of the code. There are also tools that guarantee to be sound with respect to all possible executions of the program; that is, they guarantee to report data races in the program even if they did not occur in this particular execution. Again, there is a tradeoff between the two approaches and their efficiency and usability in practice.

Similarly to soundness and precision, a tool's run-time performance and efficiency are important to consider. Even if a tool provides detailed and correct analysis of data race occurrences in a program, if it slows down program execution significantly, it is not particularly useful to a programmer trying to debug their code. Often times tools that report to be sound and precise are lacking in this aspect, which makes them difficult to use frequently and successfully.

Eraser [9] is a sound tool that uses a lockset based algorithm to detect data races and reports a slowdown of 10 to 30 times when implemented on the Digital Unix operating system on the Alpha processor, using the ATOM binary modification system. On the other hand, FastTrack [8], a vector clock based algorithm, reports a slowdown of around 10 when implemented on top of unmodified JVMs. This is comparable to Eraser's performance when implemented in the same way. FastTrack reports to be one of the fastest vector-clock based data race detectors; it outperforms other such algorithms at least by a factor of 2 [8].

FastTrack is better or as good as Eraser along all axes: it is also sound, almost as fast, and completely precise (compared to Eraser, which is very imprecise). However, while Eraser is the canonical example of lockset-based data race detection, it has a lot of shortcomings and areas of possible improvement. This work aims to improve its precision by including thread-local analysis. Combined with possible future performance optimizations, such as eliminating the overhead of checking both the thread and the lockset of a memory location based on predictive profiling work, this could lead lockset-based data race detection to maintain its soundness, significantly improve precision, and achieve performance better than FastTrack.

Chapter 3

Sound Thread-Aware Lockset Algorithms

I develop a series of algorithms that modify and enhance the basic Eraser Lockset algorithm [9]. For clarity, I refer to that algorithm as Lockset, or LS, for the rest of this work. I changed the Lockset algorithm along two dimensions: first, I modified the way in which locksets are recorded, and second, I added thread tagging to support different patterns of thread-locality in program execution.

This chapter presents the algorithms in increasing order of precision, with each algorithm making an incremental improvement in one of the two dimensions of change. I begin with a discussion of the shortcomings of the basic Lockset algorithm. Then, to illustrate the improvement in precision, each algorithm is presented with an example of a safe access pattern that Lockset and the previously described algorithms would fail to detect as safe. I also present a justification of the soundness of each modification. Table 3.1 contains a summary of the algorithms and the key change each one makes to improve precision.

3.1 Lockset Algorithm

To motivate my contributions to enhanced lockset data race detection, I first present the limitations of the Lockset (LS) algorithm. The algorithm's soundness can be justified using the happens-before relations built by the program. For any two memory accesses a and b to a location x to be considered data race-free by Lockset, there must be at least one common lock m held at each of the access. There are only two situations in which that is possible. First, it is possible that the common lock has been released after the first access and acquired before the second. This lock release-acquire synchronization sequence establishes a happens-before edge between the two memory accesses in

Algorithm		Thread-Local Check On	Update	
			LockSet	Thread
Lockset Handoff	LH	—	Current	None
Lockset Intersection Private Suffix	LI-PS	Empty lockset	Intersect	Current
Lockset Handoff Private Suffix	LH-PS	Empty lockset	Current until empty intersect	Current
Lockset Intersection Private Reset	LI-PR	Empty lockset	Nonempty intersect else current	Current
Lockset Intersection Private Handoff	LI-PH	All	Current if local else intersection	Current
Lockset Handoff Private Handoff	LH-PH	All	Current	Current

TABLE 3.1: Summary of algorithms presented in this chapter.

different threads. If the lock has not been released and re-acquired between the two actions, then it must be that the actions are both performed by the same thread, since it is not possible for more than one thread to hold a lock simultaneously. Figure 3.1 contains an example of each of these situations. Events in a single thread are necessarily ordered by happens-before, since a single thread's actions occur in program-order. Therefore, in both cases, the two memory accesses are guaranteed to be data-race free.

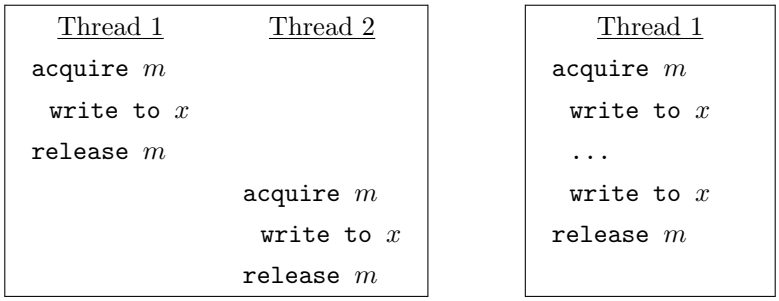

FIGURE 3.1: Memory accesses protected by the same lock. On the left, the lock is released and re-acquired between the two accesses, so they are ordered by synchronization happens before. On the right, the lock is held by a single thread that performs both accesses, so they are ordered by program order happens before.

The Lockset algorithm, though sound, is also imprecise in several ways. Firstly, it fails to detect synchronization patterns in which the lock guarding a memory location changes throughout program execution. Lockset relies on the assumption that each memory location is guarded by at least one lock consistently, and the lock must be the same throughout the entire program. However, there are patterns of program execution that do not fit this model for which Lockset reports false alarms, as shown in Figure 2.4. Another point of imprecision in the Lockset algorithm is that it is possible for shared data to become temporarily or permanently thread-local, without being guarded by a lockset, and still be free of data races because it is private. Because it lacks thread awareness,

Lockset once again reports many false positive errors in this situation. Lockset is also imprecise with respect to read-shared data, the fork-join model of synchronization, and other synchronization mechanisms. This work targets the limitations posed by the lockset recording model and the lack of thread analysis in particular.

3.2 Lockset Handoff Algorithm

Motivation The Lockset Handoff algorithm targets patterns in which the lock guarding a memory location changes throughout program execution. To illustrate this situation, consider the program trace shown in Figure 3.2. There is no data race in the trace, because there exists a happens-before edge between each consecutive pair of memory accesses. The first two accesses both share the lock m_1, since their executing threads both hold m_1 while the accesses occurs. This means that the release and acquire of m_1 establishes a happens-before relation between the first two accesses. Similarly, the second two accesses share m_2, and so the release and acquire of m_2 establishes a happens-before relation between the second and third accesses. The transitivity of the happens-before relation then establishes a happens-before edge between the first and third accesses, meaning that all of the accesses in this example are free of data races.

Thread 1	Thread 2	LS Guard State	LH Guard State
`acquire` m_1			
`write to` x		$\{m_1\}$	$\{m_1\}$
`release` m_1			
	`acquire` m_1, m_2		
	`write to` x	$\{m_1\}$	$\{m_1, m_2\}$
	`release` m_1, m_2		
`acquire` m_2			
`write to` x		$\emptyset \Rightarrow$ Error	$\{m_2\}$
`release` m_2			

FIGURE 3.2: Lockset Handoff Pattern Trace Example

The soundness of this program execution can be justified similarly to that of the Lockset algorithm. Even if there is no single lock that protects a memory location throughout the entire program execution, any two pairs of accesses to the same memory location that occur following each other must have at least one lock in common. This means that a happens-before edge is established between each pair of accesses. For an access to be considered data-race free with the previous one, it must share at least one lock with it. Similarly, the previous access must have shared at least one lock with the one preceding it. The transitivity of the happens-before relation then allows us to build an ordering of all the safe accesses in the program.

Even though the pattern shown in Figure 3.2 is safe, Lockset records the intersection of the two locksets, so it records $\{m_1\}$ as the only lock guarding x after the second access. It then reports an error on the third access, because the intersection of the current access lockset, $\{m_2\}$, and the recorded lockset, $\{m_1\}$, is empty.

Algorithm The Lockset Handoff algorithm addresses this issue by modifying the way in which locksets are recorded by Lockset. Both algorithms check that the intersection of the recorded lockset and the current lockset is not empty. Whereas the Lockset algorithm then records the intersection of the algorithm, the Lockset Handoff algorithm records the current lockset instead. That way, even if a memory location is not consistently guarded by the same lock throughout program execution, if all of the accesses to it are ordered by some lock pattern, the algorithm does not report an error. The full Lockset Handoff algorithm and state machine are presented in Figure 3.3. Each state displays the metadata recorded after a transition, which for Lockset Handoff is the lockset for this memory location. The transition arrows contain the condition for a safe transition from one memory access to another, meaning no data race exists between them.

The Lockset Handoff algorithm recognizes the trace from Figure 3.2 as a valid synchronization pattern and does not report an error. After checking the lockset intersection between each pair of accesses is not empty, it records the current lockset. Thus, the lockset recorded before the third access is $\{m_1, m_2\}$, so there is no error reported at the third access, since it shares a m_2 with the recorded lockset. This subtle difference allows Lockset Handoff to improve precision over LS.

```
1: function ACCESS(x, t)
2:     if LS(t) ∩ LS(x) = ∅ then
3:         issue a warning
4:     end if
5:     LS(x) := LS(t)
6: end function
```

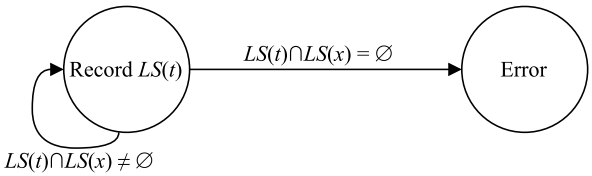

FIGURE 3.3: Lockset Handoff (LH) Algorithm and State Machine.

3.3 Thread-Aware Lockset Algorithms

As discussed in Section 2.2, any two memory accesses that are performed by the same thread are necessarily ordered by program order. No data race can occur between memory accesses from the same thread, even if there are no locks guarding the memory location. However, because they lack any thread information, both Lockset and Lockset Handoff fail to recognize unguarded thread-local memory accesses as safe. Consider the sample trace shown in Figure 3.4. The first two accesses to x share a lock, so they are data-race free. Then, even though the two accesses to x by Thread 2 are not protected by a shared lock, there is no data race between them, as they are both performed

by the same thread. However, both Lockset and Lockset Handoff report an error for that trace, because the lockset guarding x is empty at the third access

Thread 1	Thread 2	LS Guard State	LI-PS Guard State
`acquire` m_1			
`write to` x		$\{m_1\}$	$(t_1, \{m_1\})$
`release` m_1			
	`acquire` m_1		
	`write to` x	$\{m_1\}$	$(t_2, \{m_1\})$
	`release` m_1		
	`write to` x	$\emptyset \Rightarrow$ Error	$(t_2, \emptyset)$

FIGURE 3.4: Thread-Local Access Pattern Trace Example

Extending the algorithms with information that lets us compare the current access thread to the previous access thread allows us to be more precise in the performed analysis. The lockset operations performed by the Thread-Aware Lockset algorithms demonstrate the same orderings as the lockset operations in either the Eraser or Lockset Handoff algorithm, which has been established as sound already. Therefore, all pairs of memory accesses for which the Thread-Aware Lockset algorithms do not report an error are guaranteed to be free of data races because it is possible to establish a happens-before edge between them. Thus, the algorithms in their entirety are sound as well.

The Thread-Aware Lockset (TAL) algorithms all record the thread of the last access to each memory location, but they differ in the lockset computations they perform.

3.3.1 Lockset Intersection Private Suffix Algorithm

Motivation The Lockset Intersection Private Suffix (LI-PS) algorithm captures a pattern of execution in which a memory location has escaped and been accessed by more than one thread, but later becomes reprivatized and is accessed by a single thread for the rest of the program execution. This occurs, for example, when a single master thread initializes the data, then different worker threads process the data and use lockset synchronization, but eventually the data is accessed exclusively by the master thread. At that point, there is no more need for lockset synchronization, because all accesses are protected by a program-order happens before edge.

Algorithm The Lockset Intersection Private Suffix records the intersection of the current and the recorded locksets, and allows a transition if it is not empty. The one modification it makes to capture thread-locality is that if the intersection is empty, but the current access thread is the same as the recorded (i.e., last access) thread, it allows a safe transition. After such a transition, the only possible recorded lockset is the empty set, and if the memory location is ever accessed by a different

thread, the algorithm reports an error. The Lockset Intersection Private Suffix algorithm and state machine are presented in Figure 3.5. The state machine follows the same conventions as the one for Lockset Handoff, but it contains thread information as well. Transitions are based on conditions involving both. The _ symbol is used as a wild card, meaning any thread or lockset state as long as the other condition is satisfied.

```
1: function ACCESS(x, t)
2:     LS(x) := LS(t) ∩ LS(x)
3:     if LS(x) = ∅ then
4:         if t ≠ T(x) then
5:             issue a warning
6:         end if
7:     end if
8:     T(x) := t
9: end function
```

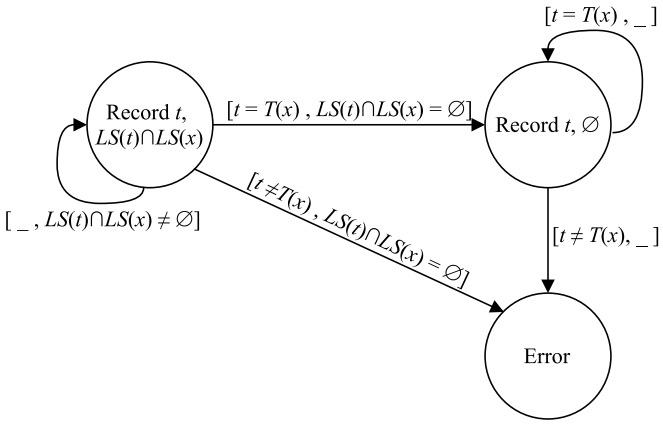

FIGURE 3.5: Lockset Intersection Private Suffix (LI-PS) Algorithm and State Machine.

3.3.2 Lockset Handoff Private Suffix Algorithm

Motivation Similarly to the pattern discussed Section 3.2, data may be guarded by more than lock throughout program execution before becoming reprivatized. For example, the trace shown in Figure 3.6 contains such an access pattern. This is still a safe access pattern, because while shared, data is protected by at least one lock in common between each pairs of accesses. Lockset Intersection Private Suffix fails to capture this pattern because it performs Lockset style computations when data is not thread-local.

Algorithm The Lockset Handoff Private Suffix (LH-PS) algorithm combines Private Suffix tracking with Lockset Handoff style lockset computations in its transitions. This means that it checks that the intersection of the current and recorded locksets is not empty, but then records the current lockset rather than the intersection. It still allows a safe transition based on an empty intersection if the current access thread is the same as the recorded thread. The full algorithm for Lockset Handoff Private Suffix is given in Figure 3.7.

Thread 1	Thread 2	LI-PS Guard State	LH-PS Guard State
`acquire` m_1			
`write to` x		$(t_1, \{m_1\})$	$(t_1, \{m_1\})$
`release` m_1			
	`acquire` m_1, m_2		
	`write to` x	$(t_2, \{m_1\})$	$(t_2, \{m_1, m_2\})$
	`release` m_1, m_2		
`acquire` m_2			
`write to` x		$(t_1, \emptyset) \Rightarrow$ Error	$(t_1, \{m_2\})$
`release` m_2			
...			
`write to` x			$(t_1, \emptyset)$

FIGURE 3.6: Lockset Handoff Private Suffix Access Pattern Trace Example

```
1:  function ACCESS(x, t)
2:      if LS(t) ∩ LS(x) = ∅ then
3:          if t ≠ T(x) then
4:              issue a warning
5:          else
6:              LS(x) := ∅
7:          end if
8:      else
9:          LS(x) := LS(t)
10:     end if
11:     T(x) := t
12: end function
```

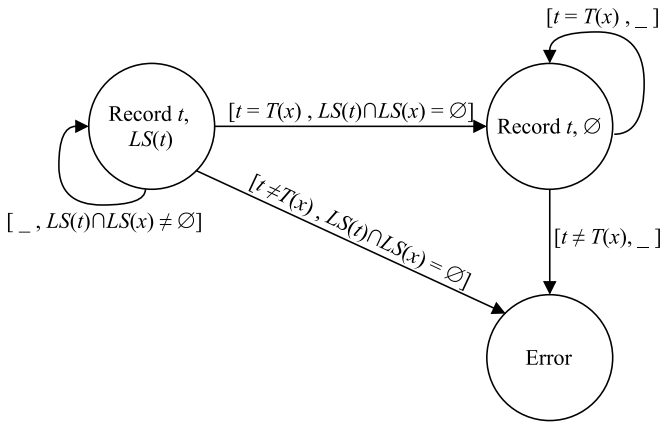

FIGURE 3.7: Lockset Handoff Private Suffix (LH-PS) Algorithm and State Machine.

3.3.3 Lockset Intersection Private Reset Algorithm

Motivation The Lockset Intersection Private Reset (LI-PR) algorithm captures patterns of execution in which a piece of data becomes temporarily thread-local and then escapes again. To illustrate such behavior, consider the trace shown in Figure 3.8. Both of the first two access occur within Thread 1, so they are ordered by program order even if they do not share a lockset. Then, the second and third accesses are both protected by m_2, so they are also safe. However, all of the algorithms presented thus far report an error for this trace. Lockset and Lockset Handoff report errors because they lack thread information which prevents them from recognizing safe program-order accesses. Lockset Intersection Private Suffix and Lockset Handoff Private Suffix report errors because they do not reset the lockset if it becomes empty, so accesses by a different thread are considered errors, regardless of whether there is a shared lock between them. Lockset Intersection

Private Reset, however, resets the lockset associated with x to $\{m_2\}$ after the second access, which allows it to transition safely to the third access.

Thread 1	Thread 2	LH-PS Guard State	LI-PR Guard State
`acquire` m_1			
`write to` x		$(t_1, \{m_1\})$	$(t_1, \{m_1\})$
`release` m_1			
`acquire` m_2			
`write to` x		$(t_1, \emptyset)$	$(t_1, \{m_2\})$
`release` m_2			
	`acquire` m_2		
	`write to` x	$(t_2, \emptyset) \Rightarrow$ Error	$(t_2, \{m_2\})$
	`release` m_2		

FIGURE 3.8: Lockset Intersection Private Reset Access Pattern Trace Example

Algorithm The Lockset Intersection Private Reset algorithm addresses that imprecision by resetting a thread-local memory location's lockset to the full set of locks held by the thread. Similarly to the other algorithms, it allows a transition over an empty lockset if the current access thread is the same as the recorded thread, but instead of keeping the empty lockset as the recorded lockset of that memory location, it sets it to be the current access lockset. As discussed above, this is a safe transition because the accesses are made by the same thread, and are thus necessarily ordered by program order. Resetting the lockset of thread-local data improves precision since it allows the memory location to be accessed later by different threads if the accesses are protected by a common lockset intersection. The Lockset Intersection Private Reset algorithm is shown in Figure 3.9.

```
1:  function ACCESS(x, t)
2:      LS(x) := LS(x) ∩ LS(t)
3:      if LS(x) = ∅ then
4:          if T(x) = t then
5:              LS(x) := LS(t)
6:          else
7:              issue a warning
8:          end if
9:      end if
10:     T(x) := t
11: end function
```

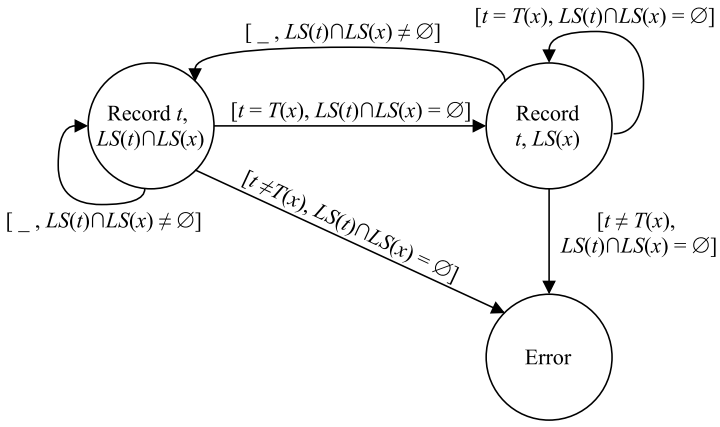

FIGURE 3.9: Lockset Intersection Private Reset (LI-PR) Algorithm and State Machine.

3.3.4 Lockset Intersection Private Handoff Algorithm

Motivation The Lockset Intersection Private Handoff (LI-PH) algorithms targets patterns like the trace shown in Figure 3.10. This trace is almost the same as the one shown in Figure 3.8, but the lockset intersection between the first and second accesses is not empty. This means that Lockset Intersection Private Reset does not reset the recorded lockset for x, which leads it to report an error on the third access.

Thread 1	Thread 2	LI-PR Guard State	LI-PH Guard State
`acquire` m_1			
`write to` x		$(t_1, \{m_1\})$	$(t_1, \{m_1\})$
`release` m_1			
`acquire` m_1, m_2			
`write to` x		$(t_1, \{m_1\})$	$(t_1, \{m_1, m_2\})$
`release` m_1, m_2			
	`acquire` m_2		
	`write to` x	$(t_2, \emptyset) \Rightarrow$ Error	$(t_2, \{m_2\})$
	`release` m_2		

FIGURE 3.10: Lockset Intersection Private Handoff Access Pattern Trace Example

```
1:  function ACCESS(x, t)
2:      if T(x) = t then
3:          LS(x) := LS(t)
4:      else
5:          LS(x) := LS(t) ∩ LS(x)
6:          if LS(x) = ∅ then
7:              issue a warning
8:          end if
9:      end if
10:     T(x) := t
11: end function
```

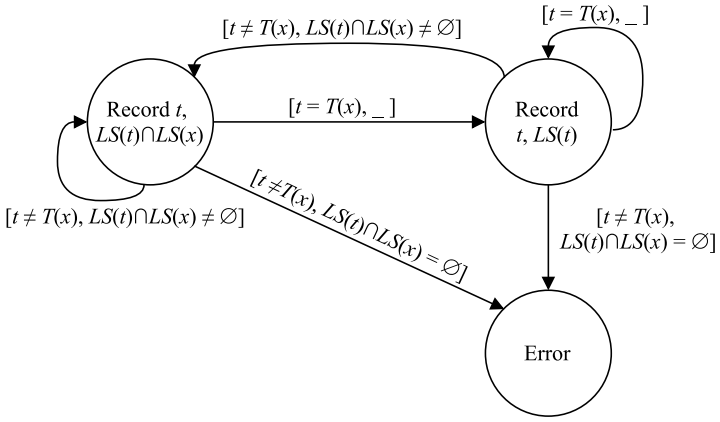

FIGURE 3.11: Lockset Intersection Private Handoff (LI-PH) Algorithm and State Machine.

Algorithm Lockset Intersection Private Handoff, on the other hand, is able to transition safely because it modifies the way in which locksets are recorded under a same-thread transition. Rather than recording the intersection of locksets or resetting it to the current lockset if the intersection becomes empty, the algorithm records the current access lockset in all thread-local accesses. If the current access thread is different than the recorded one, then the algorithms behaves as the basic

Lockset algorithm and records the intersection between the two locksets. The full algorithm for Lockset Intersection Private Handoff is in Figure 3.11.

3.3.5 Lockset Handoff Private Handoff Algorithm

Thread 1	Thread 2	LI-PH Guard State	LH-PH Guard State
`acquire` m_1			
`write to` x		$(t_1, \{m_1\})$	$(t_1, \{m_1\})$
`release` m_1			
`acquire` m_2			
`write to` x		$(t_1, \{m_2\})$	$(t_1, \{m_2\})$
`release` m_2			
	`acquire` m_2, m_3		
	`write to` x	$(t_2, \{m_2\})$	$(t_2, \{m_2, m_3\})$
	`release` m_2, m_3		
`acquire` m_3			
`write to` x		$(t_1, \emptyset) \Rightarrow$ Error	$(t_1, \{m_3\})$
`release` m_3			

FIGURE 3.12: Lockset Handoff Private Handoff Access Pattern Trace Example

Motivation The last, and most precise, variation of the Thread-Aware Lockset algorithms is the Lockset Handoff Private Handoff (LH-PH) algorithm. It captures patterns of execution in which the guarding locks of a memory location change throughout program execution, and the location becomes intermittently thread-local and then shared again. A small version of such a pattern can be seen in Figure 3.12. It presents an example program trace which is free of data races, but all other algorithms report an error on.

This trace is free of data races. The only types of accesses and transitions are those already shown to be sound in previous sections. However, as Figure 3.14 illustrates, Lockset Handoff Private Handoff is the only algorithm that successfully transitions through all four accesses.

Algorithm The Lockset Handoff Private Handoff algorithm combines the analysis of all the previous algorithms into a most precise generalized version. Its lockset recording procedure is the same as that of the Lockset Handoff algorithm, in that it always records the current access lockset regardless of thread locality. This means that the Lockset Handoff Private Handoff algorithm reports an error only when both (a) the intersection of the two locksets is empty and (b) the current and recorded threads differ. In all other cases, it considers the transition error-free. The full algorithm of Lockset Handoff Private Handoff is shown in Figure 3.13.

```
1: function ACCESS(x, t)
2:     if LS(t) ∩ LS(x) = ∅ then
3:         if T(x) ≠ t then
4:             issue a warning
5:         end if
6:     end if
7:     LS(x) := LS(t)
8:     T(x) := t
9: end function
```

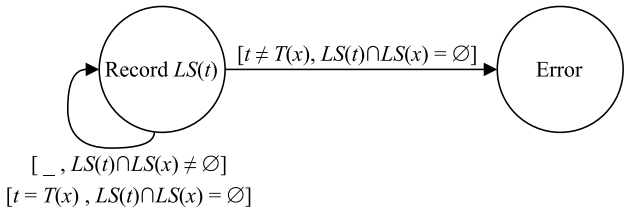

FIGURE 3.13: Lockset Handoff Private Handoff (LH-PH) Algorithm and State Machine.

Figure 3.14 illustrates the behavior of Lockset Handoff Private Handoff along with all the other algorithms, and demonstrates the improvement in precision from each algorithm. Each row represents an access to x, with the ID of the thread and the currently held lockset shown in the first two columns. All other columns contain the metadata that each algorithm records for x after the access, or "Error" if the algorithm reports an error there.

Thread	Lockset	LS	LH	LI-PS	LH-PS	LI-PR	LI-PH	LH-PH
1	$\{m_1\}$	$\{m_1\}$	$\{m_2\}$	$t_1, \{m_1\}$	$t_1, \{m_1\}$	$t_1, \{m_1\}$	$t_1, \{m_1\}$	$t_1, \{m_1\}$
1	$\{m_2\}$	Error	Error	$t_1, \emptyset$	$t_1, \emptyset$	$t_1, \{m_2\}$	$t_1, \{m_2\}$	$t_1, \{m_2\}$
2	$\{m_2, m_3\}$			Error	Error	$t_2, \{m_2\}$	$t_2, \{m_2\}$	$t_2, \{m_2, m_3\}$
1	$\{m_3\}$					Error	Error	$t_1, \{m_3\}$

FIGURE 3.14: Algorithm behavior for example access pattern shown in Figure 3.12.

Chapter 4

Implementation

I implemented the Lockset Handoff and Thread-Aware Lockset algorithms outlined in Chapter 3 for multithreaded Java programs using the RoadRunner dynamic analysis framework [13]. This chapter describes the implementation of the tools. Section 4.1 reviews relevant details about the RoadRunner framework. Sections 4.2 and 4.3 describe the implementation of the Lockset Handoff and Thread-Aware Lockset tools respectively.

4.1 RoadRunner Framework Background

The RoadRunner dynamic analysis framework [13] provides functionality to instrument multithreaded code and track metadata for memory locations and synchronization principals. The core of the RoadRunner API is presented in Figure 4.1.

RoadRunner instruments each memory location with a shadow variable to track metadata associated with that memory location. `ShadowVar` objects are stored alongside the corresponding memory location they track. For example, the Eraser implementation that comes with RoadRunner defines a `LockSet` extension of `ShadowVar` that is used to implement the Lockset algorithm, which is discussed in more detail below. The shadow data for each memory location can be accessed and modified on events such as memory access (`access` method) and initialization (`makeShadowVar` method). Threads are also instrumented with `ShadowThread` objects. By defining their own types of shadow variables and program event handlers through extending the relevant classes, analysis tools can track any information necessary for their computations. RoadRunner lets users implement the functionality of their algorithm in the form of extensions to the core `Tool` class. Each `Tool` can define instrumentation mechanisms for the events that RoadRunner monitors, such as memory accesses and lock releases.

```
    // Decorations
    class Decoratable { ... }

    class Decoration<Key extends Decoratable,Value> {
]
    }

    // Thread, Lock, and Variable Shadows
    interface ShadowVar { }

    class ShadowThread extends Decoratable implements ShadowVar {
        Thread thread;
        int tid;
        ShadowThread parent;
        static <T> Decoration<ShadowThread,T> makeDec(T init);
    }

    class ShadowLock extends Decoratable {
        Object lock;
        static <T> Decoration<ShadowLock,T> makeDec(T init)
    }

    // Events
    class Event {
        ShadowThread thread;
    }

    class AcquireEvent extends Event {
        AcquireInfo info; // loc
        ShadowLock lock;
    }

    class ReleaseEvent extends Event {
        ReleaseInfo info; // loc
        ShadowLock lock;
    }

    class AccessEvent extends Event {
        ShadowVar shadow;
        boolean putShadow(ShadowVar newShadow) { ... }
    }

    class FieldAccessEvent extends AccessEvent {
        FieldAccessInfo info; // loc, class/field desc.
        Object target; // reciever
    }

    class ArrayAccessEvent extends AccessEvent {
        ArrayAccessInfo info;
        Object target; // reciever
    }

    // Tools
    abstract class Tool {
        void create(NewThreadEvent e) { ... }
        void acquire(AcquireEvent e) { ... }
        void release(ReleaseEvent e) { ... }
        void access(AccessEvent e) { ... }
        ShadowVar makeShadowVar(AccessEvent e);
    }
```

FIGURE 4.1: RoadRunner Core API

When using RoadRunner, clients can specify which fields to instrument, and the maximum number of errors that should be reported for each field. If a field in a target program that is instrumented by a specific tool reaches the maximum number of error reports, `access` information for that field stops being passed on to the `Tool`.

4.1.1 Tool Composition

Tools can be chained together to perform more complex analysis. The first tool in the chain is used to instrument all code initially. If a field reaches the maximum number of warnings, instrumentation of that field passes onto the next tool in the chain. At that point, the second tool initializes its shadow variable for that memory location. This behavior allows tools to be used as filters for other tools to eliminate unnecessary checks at the beginning of program execution. The RoadRunner framework comes with a variety of such tools, two of which are the Thread-Local (TL) and Read-Shared (RS) filters used for Eraser and described in Section 2.4.1.1.

The Thread-Local tool records the creator thread of each memory location as the shadow data for that location. When the location is accessed, the TL tool checks if the accessor thread is the same as the creator thread, and if so, does nothing. When a memory location is accessed by a different thread, the tool advances the memory location to the next tool in the tool chain. The Read-Shared tool filters out warnings for read-shared data by verifying that accesses to the instrumented memory locations are only reads. When a write occurs, the memory location is advanced to the next tool. The use of these filters poses a risk of unsoundness, since the next tool's shadow data is not initialized until the memory location is advanced. This means that if using the TL tool, it is possible that the first access by a different thread constitutes a race with the previous access, but the successor tool would lack the necessary information to establish that. The same applies to the first write access to a memory location when using the RS tool.

The RoadRunner implementation provides a Split tool that passes all events to the two or more tools in parallel composition. Split tool chains are used to reason access by access about the difference in behavior of tools on the exact program execution.

4.2 Lockset Handoff Tool Implementation

I implented a `HandoffTool` class that extends the basic RoadRunner tool class and implements the Lockset Handoff algorithm. The basic code of the class is given in Figure 4.2, with some impertinent details abstracted away. The only metadata required for the Lockset Handoff tool is the recorded lockset for each memory location and thread. This means that memory locations are instrumented with a `LockSet` shadow variable. When a new memory location is initialized, the shadow variable is initialized to the current thread's lockset in the `makeShadowVar` method. When a location is accessed,

```
// LockSet implementation packaged with RoadRunner
class LockSet implements ShadowVar {
    static LockSet empty()
    LockSet add(ShadowLock lock)
    LockSet remove(ShadowLock lock)
    LockSet intersect(LockSet other)
    boolean isEmpty()
}

class HandoffTool extends Tool {

    static Decoration<ShadowThread, LockSet> locksHeld =
        ShadowThread.makeDec(LockSet.empty());

    void acquire(AcquireEvent e) {
        Set ls = locksHeld.get(e.thread);
        locksHeld.set(e.thread, ls.add(e.lock));
    }

    void release(ReleaseEvent e) {
        LockSet ls = locksHeld.get(e.thread);
        locksHeld.set(e.thread, ls.remove(e.lock));
    }

    ShadowVar makeShadowVar(AccessEvent e) {
     return locksHeld.get(e.thread);
    }

    void access(AccessEvent e) {
        LockSet ls = (LockSet)e.shadow;
        LockSet current = locksHeld.get(e.thread);
        LockSet inter = ls.intersect(current);
        e.putShadow(inter);
        if (inter.isEmpty()) error();
    }
}
```

FIGURE 4.2: Lockset Handoff Tool Simplified Core Implementation

the tool's `access` method is called. The `AccessEvent` object contains relevant information, such as the accessor thread and details about the memory location along with its shadow variable. The `HandoffTool` class uses that information to implement its core lockset algorithm and report an error if necessary.

To facilitate performance, `HandoffTool` uses the optimized lockset implementation packaged with the Lockset tool RoadRunner. To minimize the number of lockset allocations, the `LockSet` class memoizes each created `LockSet`, maintaining a unique data structures for each unique set of locks. Similarly to the Lockset tool, `HandoffTool` also maintains a map from thread to current lockset, which is set of locks it holds (`LocksHeld`). On lock release and acquire events, the corresponding `release` and `acquire` methods are called. They update the thread's locksets accordingly. Upon allocating a new memory location, the Lockset Handoff tool initializes that location's shadow data with the current lockset of the creator thread.

```
// Lockset Thread Pair
class LSThreadID implements ShadowVar {
    final ShadowThread thread;
    final LockSet ls;
}

class LHPHTool extends Tool {

    static Decoration<ShadowThread, LockSet> locksHeld =
        ShadowThread.makeDec(LockSet.empty());

    void acquire(AcquireEvent e) {
        Set ls = locksHeld.get(e.thread);
        locksHeld.set(e.thread, ls.add(e.lock));
    }

    void release(ReleaseEvent e) {
        LockSet ls = locksHeld.get(e.thread);
        locksHeld.set(e.thread, ls.remove(e.lock));
    }

    ShadowVar makeShadowVar(AccessEvent e) {
        return new LSThreadID(e.thread, locksHeld.get(e.thread));
    }

    void access(AccessEvent e) {
        LockSet recordedls = ((LockSet)e.shadow).ls;
        ShadowThread recordedThread = ((LockSet)e.shadow).ls;
        LockSet currentls = locksHeld.get(e.thread);
        ShadowThread currentThread = e.thread;

        LockSet interls = recordedls.intersect(currentls);

        e.putShadow(new LSThreadID(currentThread, currentls));

        if (currentThread != recordedThread && interls.isEmpty()) {
            error();
        }
    }
}
```

FIGURE 4.3: Lockset Handoff Private Handoff Tool Simplified Core Implementation

4.3 Thread-Aware Lockset Tools Implementation

I also implemented five `Tool` extension classes corresponding to each of the Thread-Aware Lockset algorithms. The implementation details for all five of the tools are almost identical, with the exception of the `access` method. As an example, the code for Lockset Handoff Private Handoff (LH-PH) tool is given below.

The Thread-Aware Lockset tools need to track both lockset and thread metadata. Therefore, I implemented an `LSThreadID` class that extends `ShadowVar` and contains a `ShadowThread` and `LockSet` object. When a memory location is initiated and `makeShadowVar` is called, a new `LSThreadID` object is created that contains the creator thread's `ShadowThread` and currently held `LockSet`. On `access` events, the object is updated as required by the respective algorithm, and errors are reported as

necessary. `ShadowThread` objects are once again associated with a lockset that is updated on lock `release` and `acquire` events.

4.3.1 Optimizations

To minimize the number of `LSThreadID` allocations, each `ShadowThread` is associated with a map from `LockSet` to `LSThreadID` containing that thread/lockset pair. When the shadow variable `LSThreadID` for a memory location needs to be updated, the tool first checks if an `LSThreadID` instance for the lockset and thread ID already exists, and if so, uses the unique object representation of the pair. Otherwise, the tool creates a new object and puts it in the map for future use. This means that the tool maintains a unique `LSThreadID` instance of each lockset and thread ID combination throughout program execution. The map uses weak references, so that if a lockset and thread ID combination is not being used as the shadow variable for any memory location, it may be collected by the garbage collector.

Chapter 5

Evaluation

To evaluate the precision and performance of the Lockset Handoff and Thread-Aware Lockset tools, I conducted a series of experiments that measure the number of errors reported and the run-time slowdown incurred by each tool for a set of benchmarks. In order to characterize the difference in the type of errors reported by each tool, I also performed detailed case studies on two benchmarks. This section describes the setup of the experiments (Section 5.1) and discusses the precision (Section 5.2) and performance (Section 5.3) results, and presents the findings of the case studies (Section 5.4).

5.1 Experimental Setup

We conducted the experiments on a machine running the Ubuntu 16.04 LTS distribution with Linux kernel version 4.4.0 on 2 × 18-core Intel Xeon E5-2695 v4 CPUs at 2.1GHz with simultaneous multithreading disabled and 256GB of RAM. The tools were implemented using RoadRunner v0.4 [13] and executed with OpenJDK 1.8.0_121 and OpenJDK Hotspot 64-Bit Server VM.

To test the tools' precision and performance, I used benchmarks from the DaCapo [17] benchmark suite. Each benchmark was run under RoadRunner with each of the tools, as well as Eraser [9] and FastTrack [8] for baseline comparison. The recorded results for each configuration are the average of 10 experimental runs.

We ran experiments on the following benchmarks from version 9.10 of the DaCapo suite:

- avrora: simulates AVR microcontrollers
- eclipse: performance tests for the Eclipse IDE
- fop: parses and formats it XSL-FO file to a PDF file

- h2 and h2small: a model of a banking application executing transactions
- jython: a Python interpreter written in Java
- luindex and lusearch: indexing and search of a corpus of text documents
- pmd: analyzes a set of Java classes for source code problems
- sunflow: renders a set of images using ray tracing
- xalan: transforms XML documents into HTML

Other benchmarks from the suite were omitted because of RoadRunner compatibility or execution issues.

5.2 Precision Evaluation

We evaluated the precision of the tools in two different ways. We ran each tool with each benchmark individually, and recorded the average number of errors reported over 10 runs, with the maximum number of warnings for each field or array declaration set to 100. This is the default RoadRunner setting, and increasing the maximum number of warnings results in significant slowdowns due to the added overhead of the error reporting mechanism. We present the observations from 10 executions. We also include runs with the Thread-Local (TL) and Read-Shared (RS) filters for Lockset, since they are part of the full Eraser algorithm. The memory locations for which errors are reported vary slightly between executions due to the fact that the tools perform dynamic analysis, the precision and soundness of which is limited to the current program execution, and are actually unsound if prefixed by the TL and RS filters as discussed in Sections 2.4.1.1 and 4.1.1. We present the results of the experiment in Table 5.1. The number of errors shown for each tool and benchmark is the average across the observed runs. There was very little variation between the runs, and the minimum and maximum number of errors reported by each tool for each benchmark were within 5% of the average.

To gain a better understanding of the precision and soundness of the tools, I ran each benchmark with FastTrack (FT) [8], Lockset prefixed with the Thread-Local and Read-Shared filters (TL:RS:LS), Lockset Handoff (LH), and the Thread-Aware Lockset (TAL) tools in a single execution, using the split tool described Section 4.1.1. For each tool, I recorded the set of fields on which errors were reported. We focused on errors for instance and static fields only because array access errors are reported by the line at which the access occurred. Therefore, it is much harder to meaningfully reason about whether two tools reported errors for the same array, since they could have reported them at different points of program execution. Since FastTrack is both precise and sound, I used the errors reported by it as a baseline. To find the number of true errors reported by a tool for a benchmark, I calculated the size of the intersection between the sets of errors FastTrack reports and the set of errors reported by the tool under evaluation. To calculate the number of false errors, I

	LockSet Analysis				Thread-Aware LocksetAnalysis					FT
	LS	TL:LS	TL:RS:LS	LH	LI-PS	LH-PS	LI-PR	LI-PH	LH-PH	
avrora	25833	10073	2732	25185	4619	4614	4224	4224	4213	300
eclipse	639789	124555	37537	620079	77854	76973	73265	72560	72519	702
fop	150999	0	0	150690	0	0	0	0	0	0
h2	52451	10200	1054	49333	11099	10428	8415	8420	8279	961
h2small	46357	9107	1052	42404	8249	7487	6008	6055	5981	431
jython	157323	148	4	156338	40	39	38	38	38	30
luindex	39639	232	9	37196	94	92	53	53	52	1
lusearch	38185	2381	0	38005	2469	2469	2370	2370	2371	901
pmd	83629	6777	120	83143	7718	7716	7685	7667	7594	2081
sunflow	37340	17902	400	35910	18720	18710	18255	18273	18185	690
xalan	48154	18438	2400	47588	18572	18473	18330	18335	18180	400

TABLE 5.1: Average number of errors reported by each tool for each benchmark, with the maximum number of warnings per memory location set to 100.

found the size of the non-symmetric set difference between the errors reported by the tool and the errors reported by FT. To calculate the number of missed errors, I found the size of the set difference between the errors reported by FT and the errors reported by the tool, since any error reported by FastTrack is a true error. The results of this experiment are recorded in Table 5.2. For each tool and benchmark, I present the number of true errors (which is the number of errors that both the tool and FastTrack reported), the number of false errors (which is the number of errors the tool reported but FastTrack did not), and the number of missed errors (which is the number of errors that FastTrack reported but the tool did not).

The Lockset Handoff (LH) tool reports the same number of errors on average as Lockset (LS), so the modification in the lockset recording algorithm does not seem to make a significant difference to the results. The benchmarks tested did not rely solely on the synchronization patterns that LH tracks. They use other types of synchronization, or have thread-local and read-shared data, in addition to any locking discipline they follow. Thus, the Lockset Handoff pattern by itself is not sufficient to capture the pattern of all memory location accesses, even if they are safe, which was an expected result given the natural limitations of the algorithm.

All of the Thread-Aware Lockset (TAL) tools report fewer errors on average than both Lockset and Lockset Handoff do, which indicates an improvement in precision. Furthermore, the results from Table 5.2 demonstrate that the Thread-Aware Lockset tools are sound, as they do not miss any errors. The difference in precision varies across benchmarks, but the tools show an average improvement of 82% fewer false errors reported compared to Lockset. In some benchmarks, such as avrora and xalan, the Thread-Aware Lockset tools do report many fewer errors than LS, they still do

		TL:RS:LS	LH	LI-PS	LH-PS	LI-PR	LI-PH	LH-PH
avrora	true	3	3	3	3	3	3	3
	false	34	310	94	93	93	92	92
	missed	0	0	0	0	0	0	0
eclipse	true	14	21	21	21	21	21	21
	false	301	4282	1127	1094	1080	1044	1044
	missed	7	0	0	0	0	0	0
fop	true	0	0	0	0	0	0	0
	false	0	1599	0	0	0	0	0
	missed	0	0	0	0	0	0	0
h2	true	1	1	1	1	1	1	1
	false	3	430	109	102	82	82	82
	missed	0	0	0	0	0	0	0
h2small	true	1	1	1	1	1	1	1
	false	3	429	107	100	81	81	81
	missed	0	0	0	0	0	0	0
jython	true	3	21	21	21	21	21	21
	false	0	969	5	5	4	4	4
	missed	18	0	0	0	0	0	0
luindex	true	1	1	1	1	1	1	1
	false	1	506	31	31	16	16	16
	missed	0	0	0	0	0	0	0
lusearch	true	0	0	0	0	0	0	0
	false	0	346	25	25	24	24	24
	missed	0	0	0	0	0	0	0
pmd	true	2	18	18	18	18	18	18
	false	0	760	17	17	16	16	16
	missed	16	0	0	0	0	0	0
sunflow	true	3	5	5	5	5	5	5
	false	1	281	129	129	125	125	125
	missed	2	0	0	0	0	0	0
xalan	true	1	6	6	6	6	6	6
	false	12	491	147	146	147	146	146
	missed	5	0	0	0	0	0	0

TABLE 5.2: The number of memory locations for which each tool reported true and false errors, and the number of missed errors for each tool and benchmark.

not compete with FastTrack's precision levels. For example, in avrora the TAL tools eliminate 82% of the false errors reported LS, but still report more than 14 times the number of errors reported by FastTrack. On benchmarks such as jython and lusearch, the number of errors reported by each of the TAL tools is comparable to that reported by FastTrack, and is a 99% improvement in precision over the Eraser and LH tools.

The full Eraser algorithm, which consists of the TL:RS:LS tool, reports significantly fewer errors than any of the Thread-Aware Lockset tools for all benchmarks (excluding fop, which has no shared data). The results in Table 5.2 also show that the tool allows actual errors to pass unreported. There are benchmarks such as avrora and lusearch for which the tool does not miss any errors, there are others such as pmd and jython for which the tool misses a significant portion of the errors reported by FastTrack. In contrast, the Thread-Aware Lockset tools report many false errors but they never miss an actual data race and report all true errors as well.

The fact that all of the Thread-Aware Lockset tool variations report almost the same number of errors is consistent with the small difference in errors reported by Lockset and Lockset Handoff, since the lockset-recording procedure does not have a significant impact on precision in practice. These results indicate that tracking the thread accessing a memory location is more useful for filtering unnecessary data race checks, while modifying the lockset algorithm to record the current lockset rather than the intersection of the recorded and current lockset has little effect on the precision of the tools for these applications This is not surprising because a lot of program data is thread-local, or becomes local to a single thread after being initialized by a main thread. It is possible that the recording style of Lockset Handoff makes a difference over LS only when combined with the Private Handoff analysis. However, this is probably rare and would be hard to measure in practice.

The improvement in precision is also due to the fact that even if data is shared by two or more threads without synchronization, the TAL tools report an error only on the first access by a different thread. All subsequent accesses by the same thread are considered safe, and no error is reported for them. Thus, volume of errors reported for each memory location is decreased. Furthermore, while there are cases in which LI-PS reports more false alarms than the other TAL variations, the comparable precision of all these tools would suggest that patterns of reprivatization at the end of data lifetime are relatively more common compared to intermittent thread-locality.

5.3 Performance Evaluation

To evaluate performance, I conducted a series of experiments that recorded the average running time of each benchmark instrumented with each tool. Figure 5.1 and Table 5.3 show the performance experiment results. We present the average slowdown of each tool for each benchmark, normalized to the benchmark's native runtime.

	LockSet Analysis				Thread-Aware LocksetAnalysis					FT
	LS	TL:LS	TL:RS:LS	LH	LI-PS	LH-PS	LI-PR	LI-PH	LH-PH	
avrora	3.3	3.3	3.2	3.3	7.1	7.0	7.8	7.0	7.6	4.0
eclipse	13.2	12.5	12.3	13.2	16.1	16.6	17.2	16.4	16.3	14.8
fop	8.4	4.8	4.8	8.4	5.5	5.6	5.4	5.4	5.4	5.8
h2	13.9	12.6	11.2	13.4	19.0	18.1	18.8	18.4	18.2	13.8
h2small	13.7	11.5	11.1	13.2	19.0	19.4	19.8	17.7	18.3	13.5
jython	7.7	6.5	6.4	7.7	7.9	8.0	8.1	8.0	7.9	7.5
luindex	7.8	5.7	5.7	7.7	11.9	11.7	12.2	11.3	11.8	9.0
lusearch	6.7	6.2	6.2	6.7	8.7	8.4	7.6	7.9	7.3	7.6
pmd	7.7	6.9	6.3	7.7	7.8	7.7	7.9	7.5	7.6	8.0
sunflow	10.1	9.0	8.3	10.3	15.9	16.9	15.5	16.6	15.1	11.7
xalan	8.0	5.6	7.9	6.4	7.7	6.1	9.1	6.0	6.1	6.7

TABLE 5.3: Average run time slowdown incurred by each tool for each benchmark normalized to the benchmark's native performance, with the maximum number of warnings per memory location set to 100.

The Lockset Handoff tool's performance is comparable to that of Lockset, which is expected given the similarity in their logic and implementation. Both tools record the same type of metadata and perform very similar computations in the instrumented code. They also both used the memoized lockset implementation described in Section 4.3.

The series of Thread-Aware Lockset (TAL) tools all perform comparably to each other, but on average incur a slowdown of 1.17 times that of Lockset. For some of the benchmarks, the tools incur a significant slowdown compared to Lockset and FastTrack. This is most evident in avrora, on which the Thread-Aware Lockset tools incur a slowdown of as much as 7 times relative to the benchmark's native performance. This is more than twice the slowdown of Lockset and FastTrack incur on the same benchmark. Other benchmarks for which TAL tools perform significantly worse than the simple LockSet tools are eclipse, h2, h2small, and luindex. On all of these, the TAL tools incur a slowdown that is 1.3-1.5 times that of LS and FT.

The Thread-Aware Lockset tools perform as well as or better than Lockset on several of the benchmarks. This is most notable in fop, for which the TAL tools have a 36% improvement in performance compared to LS. This is due to the fact that the fop application performs no thread-sharing of data. A similar result is shown in jython, which has very little shared data. This is likely due to the fact that the Thread-Aware Lockset tools report significantly fewer errors for these benchmarks, and error reporting is expensive in RoadRunner. For xalan, lusearch and pmd, the performance is mostly comparable across all of the tested tools, while precision on these benchmarks is significantly improved over Lockset's precision.

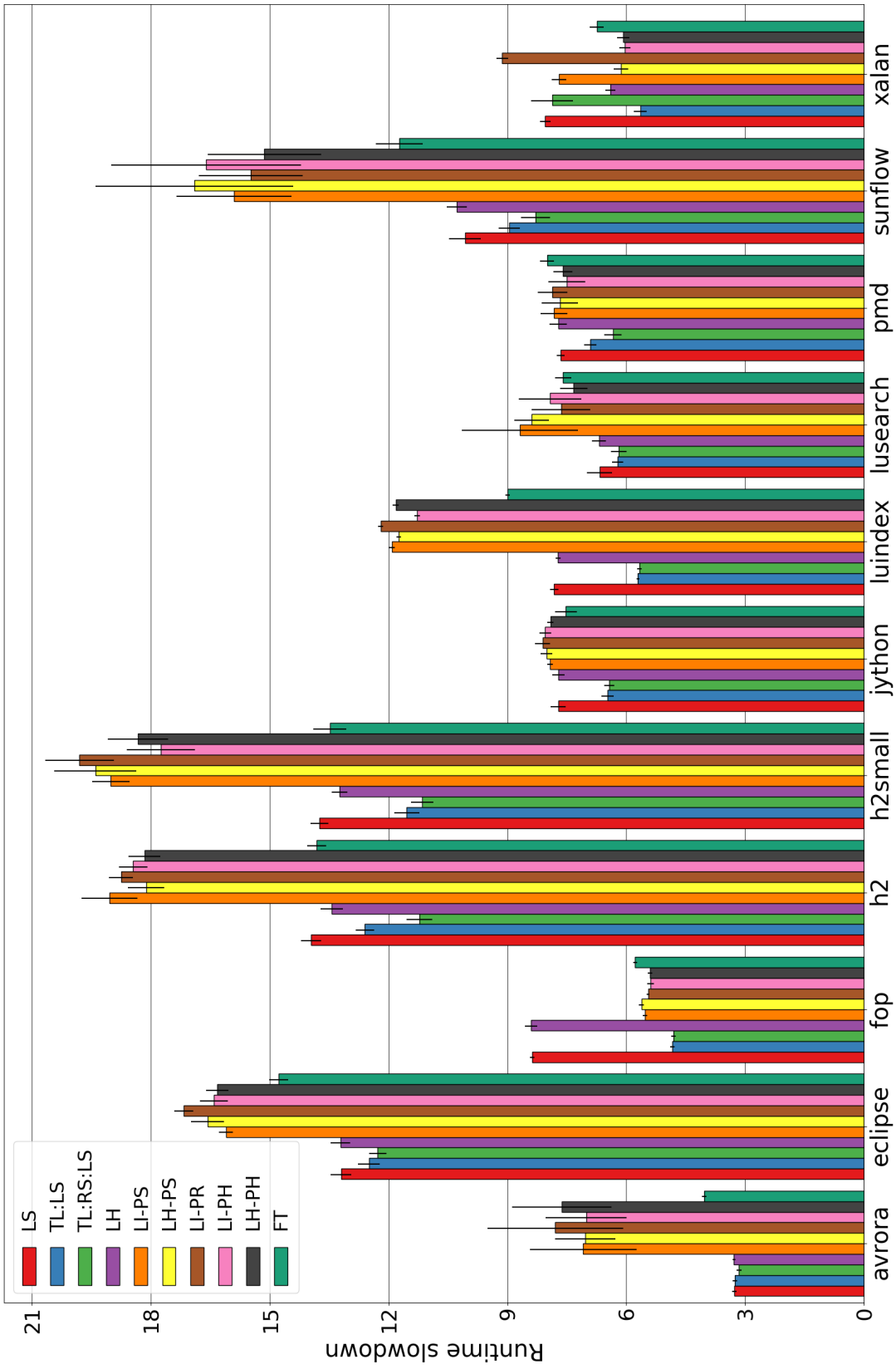

FIGURE 5.1: Average slowdown for each tool for each benchmark.

5.4 Case Studies

To better understand the types of synchronization and program execution patterns that the Thread-Aware Lockset algorithm tools capture, I examined the errors reported by different tools for avrora and xalan. We chose these benchmarks because the tools' comparative behavior is different for these two benchmarks, and it was possible to find the source code of each of them. Since Lockset Handoff Private Handoff is the most precise of the tools, I used that tool in the analysis. We used the errors reported by Eraser as a baseline for the false positive errors that Lockset Handoff Private Handoff eliminated. Similarly, I used FastTrack as a precision baseline, with the assumption that any error reported by FastTrack is a true data race. We also ran the tools with combinations of the Thread-Local (TL) and Read-Shared (RS) filters described in Section 4.1.1 to determine how much improvement the augmented Thread-Aware Lockset analysis provides over the full Eraser algorithm, including its use of Thread-Local and Read-Shared filtering. This allows us to reason about what the more sophisticated thread-local tracking contributes over the simple Thread-Local filter in Eraser.

We ran the case study experiments using the Split tool (Section 4.1.1), which runs the tools in parallel in the same execution. This makes it possible to reason about the exact difference in behavior between the tools. We ran the experiment with a maximum of 1 warning reported for each field or array declaration. With this setting, each tool can report a maximum of one error for each unique memory declaration before it stops tracking it. This is different from reporting a maximum of one warning per memory location, as it is possible that different instances of the same object all have a data race on a specific field, but only the first such instance would be reported. This allowed us to better distinguish between the unique memory declarations for which errors are reported by each tool. To gain an understanding for the total number of errors reported by each tool, I ran the same experiments with a maximum of 100 warnings for each memory location, which is the default RoadRunner configuration. The experiments also included filtering by the Thread-Local and Read-Shared tools. The ':' notation used between tools means that the tools are used in succession as filters (Section 4.1.1). For example, 'TL:LH-PH' means that the LH-PH tool was preceded by the Thread-Local prefix filter, and 'TL:RS:LH-PH' means that the LH-PH tool was preceded by the TL filter followed by the RS filter.

5.4.1 Avrora

The avrora benchmarks simulates programs run on a grid of AVR microcontrollers. It exhibits various data sharing patterns, and relies on primarily locks as its explicit synchronization mechanism. The average number of errors observed from the avrora case study experiments are recorded in Table 5.4.

	Number of errors reported	
Tool	1 Warning per Field	100 Warnings per Field
FastTrack	3	300
TL:RS:LH-PH	3	300
TL:RS:LS	37	2800
TL:LH-PH	39	2500
TL:LS	120	10000
LH-PH	134	4212
LS	657	25800

TABLE 5.4: Average number of errors reported for avrora by each tool with the specified maximum number of warnings.

5.4.1.1 True Data Races

To understand the behavior of each tool, it is necessary to examine the true data races in the program, which are the three unique errors reported by FastTrack. They occur on the `lastBit` field in `avrora/sim/radio/Medium$Transmission` class, and the `Pr` and `Pn` fields of the `avrora/sim/radio/Medium` class. The error on the `lastBit` field stems from the fact that the field is guarded by a lock, but inconsistently. Most uses of the fields are protected by the lock guarding the `Medium` object the `Transmission` belongs to. However, when a thread writes to the field in the `end` method of the `Transmission` class, it is accessed without lock synchronization. Tracing through the call hierarchy of the `end` method shows that the method itself is also invoked without synchronization. Thus, the write access to the `lastBit` in that method is in fact a data race.

The two other fields for which data races are reported, `Pn` and `Pr`, are both in the `Method` class and get used in the same pattern. They are modified in the `isChannelClear` method of the `Receiver` class and in the `deliverByte` method of the Ticker class, which is internal to the `avrora/sim/radio/Medium$Receiver` class, in which they are accessed without synchronization. This means that there is in fact a data race for both of these locations, which all the tools correctly detect.

5.4.1.2 False Positive Errors

The tools report errors on sets of memory locations that mostly overlap, but are not identical. TL:RS:LH-PH, which is the LH-PH tool prefixed by the Thread-Local and Read-Shared filters, reports exactly the three actual errors in the program. this suggests that the remaining memory locations for which errors are reported by the other tools are either thread-local or read-shared, and thus data race free.

It is interesting to observe that the TL:LH-PH combination reports significantly fewer errors than LH-PH alone. The cause of this difference in error counts is the transition between tools in RoadRunner's tool chain. The Thread-Local filter only advances a memory location if it is accessed by a thread other than the one that initialized it, as described in Section 4.1.1. This means that it filters out all the accesses by the creator thread. The first access for which the successor tool starts tracking is the first access by a different thread, at which point the metadata for the memory location is initialized. Therefore, if a memory location is initialized in the main program thread, and then that object is only ever accessed by a single other thread, the TL filter would prevent the LH-PH tool from reporting an error for that memory location, even if there is a true data race. Although this can be a potential source of unsoundness in the tool's implementation, this did not cause missed data races in practice in the observed executions of avrora. This is evidenced by the fact that in each experiment, FT and TL:RS:LH-PH reported the same errors, so all accesses filtered out by the TL prefix to the LH-PH tool were in fact safe.

The avrora application has many memory locations that follow such a pattern. For example, the `nesting` field in the `avrora/sim/util/TransactionalList` class is initialized when the object is created in the main thread, but the instance is then only accessed by a different thread forked after its initialization. The forking of the second thread establishes a happens-before edge between the initialization in the main thread and all subsequent access by the second thread. FastTrack detects the fork and does not report an error for the location. LH-PH does report an error, because it does not track fork and join behavior, but TL:LH-PH does not report an error. This is due to the fact that applying the Thread-Local filter means that LH-PH only tracks the accesses beginning with the second thread. However, since the new thread both reads from and writes to the field, the TL:RS filter is not enough to prevent Lockset from reporting an error for that field. There are 34 other fields in avrora that have the same behavior, and for which TL:RS:LS incorrectly reports an error, but TL:LH-PH does not.

5.4.2 Xalan

Xalan transforms XML documents into html format. The conducted studies show that it exhibits synchronization and data access patterns distinct from those of avrora, but it still contains a lot of thread-local and read-shared data. The average number of errors observed from the xalan case study experiments are recorded in Table 5.5.

5.4.2.1 True Data Races

Unlike in avrora, where FastTrack consistently reported the same three data races, the results for xalan exhibit more variance in terms of the errors reported. Across executions FastTrack reported an average of 13 errors, but there were executions with as few as 9 and as many as 19 errors reported.

	Number of errors reported	
Tool	1 Warning per Field	100 Warnings per Field
FastTrack	13	384
TL:RS:LH-PH	13	1300
TL:RS:LS	13	1300
TL:LH-PH	160	16000
TL:LS	160	16200
LH-PH	185	18184
LS	803	48152

TABLE 5.5: Average number of errors reported for xalan by each tool with the specified maximum number of warnings.

We picked a sample execution for which FastTrack reports 13 errors, which are the declarations it reports most commonly across all executions. As an example, three of these errors come from the `org/apache/xml/serializer/CharInfo` class. The `onlyQuotAmpLtGt`, `m`, and `firstWordNotUsed` fields of the `CharInfo` class are all accessed in the `mutableCopyOf` method of the class without synchronization. Errors on these memory locations are reported by FastTrack in all executions. Most of the other errors reported by FT are also in the `org/apache/xml/serializer` package.

5.4.2.2 False Positive and Missed Errors

Compared to their precision for avrora, the Thread-Aware Lockset tools provide less improvement over Lockset on xalan. Even though LH-PH reduces the number of errors reported by 4 times compared to LS, it still reports significantly more errors than FT. This is due to the fact xalan has a lot of data that is read-shared, and LH-PH is not able to detect that pattern. For example, the `m_defaultRule` field in the `org/apache/xalan/templates/StylesheetRoot` class is a read-shared field. It is only initialized once in the `StylesheetRoot` constructor, and then only accessed through the `getDefaultRule` method. After initialization, only reads occur to this default template.

The other commonly occurring sharing pattern is thread-local data that is initialized in the main thread and then only ever accessed by a single other thread forked from the main thread. The `m_firstWalker` field in the `org/apache/xpath/axes/WalkingInterator` class is an example of such a field. It is initialized in the main thread, and then only accessed by a single other thread throughout the rest of program execution. FastTrack detects the happens-before edge introduced by forking the second thread, but the Lockset Handoff Private Handoff and Lockset tools' analyses cannot detect it, thus they both report a false error for that field.

However, running both LS and LH-PH with the Thread-Local and Read-Shared filters poses another problem: it makes the analysis of the successor tools unsound, as discussed in Section 2.4.1.1. Even

though FastTrack, TL:RS:LH-PH and TL:RS:LS all report errors on 13 unique memory locations, the locations reported by TL:RS:LH-PH and TL:RS:LS are different from the ones reported by FT. Since FastTrack is both sound and precise, this means that it reports only true data races, and thus any errors reported by FT but not by the other tools are races missed by TL:RS:LS. This is due to the unsound filtering by TL and RS. For example, TL:RS:LS report none of the fields discussed in the previous section (`onlyQuotAmpLtGt`, `m`, and `firstWordNotUsed`). This example of unsoundness illustrates the compromise that the filtered Lockset algorithm makes to improve precision. In contrast, even though the Thread-Aware Lockset tools do not achieve the improved precision levels of TL:RS:LS, they never fail to report a true data race.

5.5 Discussion

Results from the precision experiments show that tracking thread information can improve precision significantly over the basic lockset algorithm. However, this improvement is dependent upon the type of access pattern that the program exhibits. If most of the data is indeed shared, or even read-shared, thread tracking does not provide significant benefits. Nonetheless, since applications commonly use at least some, and often a large amount of, thread-local data, the Thread-Aware Lockset tools improve the precision of lockset-based data race detection by an average of 82%, and as much as 99% for some benchmarks. These precision results are comparable, and in some cases better, to that of the canonical thread-local filtering optimization for Lockset. However, while the Thread-Local filter allows unsound transitions that result in the missed data races, the TAL tools are completely sound.

While the Thread-Aware Lockset tools offer significant benefits over Lockset, they are naturally limited in their precision, as they do not track synchronization other than locksets and thread locality. Furthermore, they do not target read-shared data accesses, which is a common pattern in many application. This means that they cannot achieve the levels of precision that FastTrack and other non-lockset based dynamic data race detection tools can.

The Thread-Aware Lockset tools can incur a moderate performance slowdown compared to other dynamic tools. This is partially due to the fact that they need to record more information that leads to more expensive computations and allocations for each memory access. The average slowdown relative to Lockset is 17%. For all benchmark, the Thread-Aware Lockset tools are no more than 2× slower than Lockset. Furthermore, on four benchmarks the performance of the Thread-Aware Lockset tools is competitive with that of Lockset and FastTrack, and sometimes even better, especially if the majority of data in the program is thread-local.

Overall, the Thread-Aware Lockset tools offer a sound alternative to the Thread-Local filter for Lockset. These tools provide significant precision gains at little cost to run-time performance.

Chapter 6

Future Work

6.1 Read-Sharing Analysis

Currently, the Thread-Aware Lockset tools do not recognize patterns of execution in which data is read-shared. Recognizing such patterns as safe could lead to significant increases in precision, as read-shared data is common. Tools such as Eraser's Read-Shared filter capture read-shared patterns, but introduce unsoundness when transitioning from the read-shared state to the next state. Other tools, such as FastTrack and GoldiLocks, record enough program information to recognize when data is read-shared and free of data races. It is possible there exists an intermediate solution between the two approaches that can improve the precision of the Thread-Aware Lockset tools without compromising their soundness, and without the need to track all of the additional data that FastTrack and GoldiLocks do. An example of such a solution is implementing an approach analogous to the Read-Shared filter, but as a suffix rather than a prefix. This could be used similarly to the way in which the Lockset Intersection Private Suffix and Lockset Handoff Private Suffix can correctly detect thread locality at the end of a datum's shared lifetime.

However, such a Read-Shared extension would have to address the tradeoffs between soundness and precision as well. While the Eraser Read-Shared filter is unsound because there is no way to know the exact point in program execution when data transitions from read-shared to shared-modified, any Read-Shared suffix tool would need to decide when to transition data from shared-modified to read-shared. The two simple approaches to this decision fall on either end of the precision-soundness dichotomy. We can guarantee soundness by declaring any write after the first write to a memory location an error, forcing all data to be read-shared. That would be far too imprecise. It is possible to track only the previous write and read, and allow any new read access that does not conflict with the previous write, as well as any new write access that does not conflict with the previous read. This would be imprecise because it could miss reads that occurred after the previous write but

before the previous read, and were in a data race with the current write. The limitations of both of these approaches are apparent. Future work in this area should explore the tradeoffs in this type of analysis.

6.2 Thread-Aware Lockset Optimizations

The implementation of the Thread-Aware Lockset tools evaluated in this work incurs a significant run time overhead for some benchmarks. Even though I optimized the memory allocations necessary by maintaining memoized instances of previous thread-lockset pairs, the table lookup operations required to access those instances also contribute to the slower performance. More detailed profiling of the number of allocations and table lookups would be a good first step in improving the run-time performance of the tools. It is possible to implement the thread-lockset pair objects in a way that allows each memory location to be associated with a single mutable lockset-thread instance throughout its lifetime. Although that would require additional synchronization when mutating the fields of the instance, it may prove to be less computationally expensive expensive and thus improve run time performance. Advances in this area have the potential to bring Thread-Aware Lockset performance closer to that of Lockset and FastTrack.

6.3 Combined Analysis

Even though there is little published research in the area, work has been done to profile the types of locking behavior programs follow. Preliminary results indicate that programs usually use relatively few locksets overall, the locksets are relatively small, typically one lock ([18]). The profiling work shows that it is possible to predict the lockset that a memory location will have ahead of time. This could save the time and space overhead of maintaining the lockset by replacing it with a cheap prediction verification step in most cases. The predictive power might make it possible to know exactly when thread-locality matters, which can mitigate the performance overhead typically incurred by the algorithm, leading to performance optimizations. Eventually, the combined predictive analysis may lead to a sound, more precise, and faster lockset-based data race detection algorithm that might compete with or beat FastTrack.

Chapter 7

Conclusion

This work developed a series of Thread-Aware Lockset algorithms that combine thread-local and lockset-based dynamic data race detection analysis to improve the precision and soundness of existing lockset tools. The algorithms can detect thread-local patterns of data sharing without compromising the soundness of their analysis. This is an important improvement over the existing approaches to thread-local filtering for lockset tools.

To evaluate the precision and performance of the algorithms, I implemented them for multithreaded Java programs. The evaluation indicates that the presence of thread-tracking information leads to a significant improvement in precision for most target applications. The most precise tool achieves an average improvement in precision of 82% over purely lockset-based algorithms. The significant precision improvements of the Thread-Aware Lockset tools come at a run-time slowdown of only 17% over the best lockset-based dynamic data race detection tools.

The performance and precision of the tools do not yet compete with that of vector-clock based algorithms like FastTrack, but the tools offer a fully sound and significantly more precise alternative to the standard lockset algorithms at little additional performance cost. Overall, thread-local analysis provides a non-trivial improvement in the precision of lockset-based data race detection. Combined with other possible optimizations described in Section 6.3, this can eventually lead to significant improvements to all aspects of lockset-based data race detection.

Bibliography

[1] Cormac Flanagan and Stephen N. Freund. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. *Science of Computer Programming*, 71(2):89–109, 2008.

[2] A. J. Bernstein. Analysis of Programs for Parallel Processing. *IEEE Transactions on Electronic Computers*, EC-15(5):757–763, 1966.

[3] Robert H. B. Netzer and Barton P. Miller. What Are Race Conditions?: Some Isues and Formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, 1992.

[4] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[5] Martin Abadi, Cormac Flanagan, and Stephen N. Freund. Types for Safe Locking: Static Race Detection for Java. *ACM Transactions on Programming Languages and Systems*, 28(2):207–255, 2006.

[6] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006.

[7] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. In *Proceedings of the Workshop on Parallel and Distributed Algorithms*, 1988.

[8] Cormac Flanagan and Stephen N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. *Communications of the ACM*, 53(11):93–101, 2010.

[9] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[10] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: A Race and Transaction-aware Java Runtime. *SIGPLAN Notices*, 42(6):245–255, 2007.

[11] Xinwei Xie and Jingling Xue. Acculock: Accurate and Efficient Detection of Data Races. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2011.

[12] Mark Christiaens and Koen De Bosschere. TRaDe, a Topological Approach to On-the-fly Race Detection in Java Programs. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, 2001.

[13] Cormac Flanagan and Stephen N. Freund. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2010.

[14] Yao Qi, Raja Das, Zhi Da Luo, and Martin Trotter. MulticoreSDK: A Practical and Efficient Data Race Detector for Real-world Applications. In *Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, 2009.

[15] Eli Pozniansky and Assaf Schuster. Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs. *SIGPLAN Notices*, 38(10):179–190, 2003.

[16] Paul Sack, Brian E. Bliss, Zhiqiang Ma, Paul Petersen, and Josep Torrellas. Accurate and Efficient Filtering for the Intel Thread Checker Race Detector. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, 2006.

[17] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, 2006.

[18] Kasey Shen. Profiling Synchronization Patterns in Multithreaded Programs. In *Student Posters, Consortium for Computing Science in Colleges Northeastern Conference*, 2016.