Scaling Model Checking of Dataraces Using Dynamic Information

Ohad Shacham
School of Computer Science, Tel-Aviv University, Israel

September 2004

Abstract

Dataraces in multithreaded programs often indicate severe bugs and can cause unexpected behaviors when different thread interleavings are executed. Because dataraces are a cause for concern, many works have dealt with the problem of detecting them. Works based on dynamic techniques either report errors only for dataraces that occur in the current interleaving, which limits their usefulness, or produce many spurious dataraces. Works based on model checking search exhaustively for dataraces and thus can reveal even those that occur in rarely executed paths. However, the applicability of model checking is limited because the large number of thread interleavings in realistic multithreaded programs causes state space explosion. In this work, we combine the two techniques in a hybrid scheme which overcomes these difficulties and enjoys the advantages of both worlds. Our experimental results show that our hybrid technique succeeds in providing thread interleavings that prove the existence of dataraces in realistic programs. The programs that we have experimented with cannot be checked using either an ordinary industrial strength model checker or bounded model checking.

Acknowledgments

Foremost, I would like to thank my advisers, Dr. Mooly Sagiv and Prof. Assaf Schuster, for their guidance and support during this research.

Special thanks to Sharon Barner for her immense support in using the Wolf model checker and for many helpful discussions.
I thank Dr. John Deok Choi and Dr. Robert O'Callahan for allowing us to use their Shrike tool.
I thank Dr. Christoph von Praun and Konstantin Shagin for their help with the examples.

I thank Dr. Ofer Strichman and Prof. Amiram Yehudai for their helpful comments.

I thank the Israeli Academy of Science for partial financial support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Writing multithreaded programs is known to be error prone. Dataraces in multithreaded applications occur when a thread writes into a memory location while another thread is reading from or writing into it at the same time. Dataraces often indicate severe bugs and can cause unexpected behaviors when different thread interleavings are executed. Most dataraces are caused by improper use of synchronization operations, though some dataraces are, of course, intentional. However, we argue that it is important to be aware even of those intentional dataraces inserted into certain programs. A further problem is that of programmers who try to deal with dataraces by inserting redundant synchronization operations that degrade performance or even cause deadlocks.

That dataraces can occur on rarely executed paths makes their detection difficult. In general, datarace detection for arbitrary programs is undecidable [3], and even for a restricted set of programs, such as those without branching, datarace detection is NP-complete [31]. Because dataraces are

a cause for concern, many works have dealt with the problem of detecting them. Most works rely on static [8, 12, 33] or dynamic [19, 23, 26] analysis to find dataraces, while others use model checking [9], or a combination of techniques [7, 15] for better results.

Static analysis tools such as [8, 12, 33] can check whether a program is datarace free. However, it is not clear how to apply these methods to large and complicated programs without producing spurious dataraces. Whereas dynamic analysis tools are more precise than their static counterparts, these tools can still produce spurious warnings and can report errors only for dataraces that occur in the current interleaving. This limits their usefulness because dataraces are hard to reproduce. One such dynamic analysis tool is Djit [19], which is based on Lamport's happens-before partial-order relation and uses *time vectors* [20].

Another dynamic datarace detection tool is Lockset. This tool, which is implemented in Eraser [26], is based on the assumption that well-behaved programs preserve a locking discipline. This means that, for every shared memory location, there exists a lock such that every access to the location is guarded by this lock. Thus, the algorithm guarantees the absence of dataraces in a given execution by checking that, for every shared memory location, there exists such a lock. One of the interesting strengths of Lockset is that violations of a locking discipline often indicate dataraces in different thread interleavings caused by scheduling. This helps Lockset to predict dataraces in rare execution paths and not just find errors in the current execution. However, Lockset only finds breaks of the locking discipline and not dataraces. This feature has two serious disadvantages:

1. Violation of the locking discipline does not guarantee the existence of a datarace. This causes many spurious warnings even for small and simple programs.

2. Lockset cannot provide a witness for a datarace. This makes it hard to analyze and locate the actual thread interleaving that causes the datarace.

Model checking, though considered a promising method for datarace detection, also suffers from disadvantages. Model checking searches exhaustively for dataraces and thus can reveal even those that occur in rarely executed paths. However, the applicability of model checking is limited because the large number of thread interleavings in realistic multithreaded programs causes state space explosion. Abstraction-refinement techniques can also be applied, but they do not solve the problem. Nonetheless, these techniques are useful for proving the absence of dataraces in multithreaded programs, and thus they complement our own work.

1.1 Main results

In this work we combine model checking and Lockset in order to overcome their respective difficulties. Our hybrid technique enjoys the benefits of both worlds. It provides a thread interleaving which proves the existence of a datarace between two access events in realistic programs. More precisely, *a witness* for a datarace is a program trace (Π) with an access event a_1 by a thread t_1 and an access event a_2 by a different thread t_2 to the same memory location m, such that the following conditions are met:

1. a_1 or a_2 are writing into m.

2. a_2 is the first action after a_1 on Π.

3. a_1 or a_2 are not protected access events.

We define a protected access event as an access to a memory location that takes part in a synchronization operation. This synchronization operation can be either defined by the hardware, such as Test&Set, or by the programming language, such as `synchronized` operation in Java. Our witness definition reveals a datarace by virtue of the fact that a_1 can actually be postponed until after a_2 is executed. A witness exists in a program if and only if a datarace exists in the same program.

Our algorithm constructs a witness for a datarace in two phases: First, we run the Lockset algorithm in order to produce violations of the locking discipline together with the executed trace. This trace need not be a witness. Furthermore, a violation of the locking discipline might occur even though a witness for a datarace does not exist. In the second phase, we use a model checker [2] to construct a witness trace that shares a prefix with the actual trace executed by Lockset. We exploit the violation information in order to help the model checker find a witness for a datarace. In other words, the information from the Lockset algorithm reduces the number of interleavings that the model checker needs to explore.

1.1.1 Empirical results

We have implemented a simple prototype of our algorithm and used it to generate datarace witnesses for realistic programs.

This prototype already generates witnesses for public domain Java programs. These witnesses are nontrivial and we are not aware of other tools that can produce such witnesses. In addition, the improvement of formal verification tools will only increase the usefulness of our method.

The transition system in the programs that we checked was huge, sometimes beyond the scope of our software tuned model checker. We also tried to employ a bounded model checker [4, 10], but we still failed to find witnesses in our programs using these two techniques. Our hybrid method, however, did help the model checker to handle these programs by using Lockset information to generate smaller and simpler transition systems. Our method is not limited to symbolic model checking and can use other techniques such as explicit [17] or bounded model checking.

We enhanced one of the realistic programs by adding a synchronization operation and creating a datarace which appears only when a rare interleaving is executed. This was done to demonstrate the usefulness of our technique. Because dataraces in the modified programs occur on rarely executed paths, they are hard to find. Dynamic tools such as Djit (see Section 4.2) miss such dataraces when executing different traces. As expected, Lockset finds violations of the locking discipline even by executing the programs on other traces, and the model checker produces witnesses on all these enhanced programs using Lockset information.

Chapter 2

Preliminaries

In this chapter, we formally define the notion of a datarace and describe a dynamic datarace detection tool, called Lockset, that checks for violations of locking discipline in multithreaded programs.

2.1 Memory models

In this work we assume that the memory model used by the multithreaded program is sequential consistency [21], which means that the result of every execution is exactly what it would have been had the operations of all the processes been executed in some sequential order, and that the operations of each process appear exactly as was specified in the program. We use this assumption while building the models of our multithreaded programs. In addition, due to this assumption we know that there exists a serialization of operations in every execution of our programs.

This assumption, which is commonly used by many works for model

checking of multithreaded programs [6, 14, 30], is problematic because several memory models—Java, for example—are weaker than sequential consistency. Several works suggested formalization of weak memory consistency models. Roychoudhury and Mitra [25] suggested a formal specification for the Java memory model (JMM) using guarded commands, and apply their specification when analyzing multithreaded Java programs using the Murφ model checker. Their work is focused on JMM, but it can be tuned to other weak memory consistency models. Theoretically, our work can be extended to support their formal specification to JMM. However, weak memory consistency increases the number of thread interleavings that the model checker needs to explore. This in turn increases the cost of model checking and decreases its applicability to realistic programs.

2.2 Datarace

In order to provide a formal definition of a datarace, we first give several auxiliary definitions.

A multithreaded program has a heap and a set of global variables which can be seen by all the program threads. In addition, each thread in a program has its own ID, its own local variables, and a program counter.

Definition 2.2.1 *A* **global program state** *in a multithreaded program is a tuple which contains:*

- *A program counter value for each thread.*
- *Values of all the global variables.*

- *Values of local variables for each thread.*
- *The content of the heap.*

Σ_0 denotes the set of a program's initial states.

Definition 2.2.2 *A tuple of two global program states σ, σ' and an action ac are in a* **relation** *R ($(\sigma, ac, \sigma') \in R$) if the multithreaded program can step from σ to σ' by performing the action ac.*

Definition 2.2.3 *A* **trace** *is a serialization of global program states connected by actions, such that every two consecutive states σ, σ' and their connecting action ac are in R. More formally, $\pi = [\sigma_0, ac_0, \sigma_1, ac_1, \sigma_2, \ldots,]$ s.t. $\forall i. \sigma_i, ac_i, \sigma_{i+1} \in \pi \rightarrow (\sigma_i, ac_i, \sigma_{i+1}) \in R$. A trace π is a* **program trace** *if the first state in π is in Σ_0.*

We denote by $\pi.a$, a concatenation of a program trace π and an action a. And by $\pi'.\sigma$, a concatenation of a program trace π' and a global program state σ.

Definition 2.2.4 *A program state σ is a* **reachable program state** *if there exists a program trace π such that $\sigma \in \pi$.*

Definition 2.2.5 *An* **access event** $\mathtt{a}$ *is a read operation from a memory location (m^a) or a write operation to a memory location (m^a).*

Definition 2.2.6 *An access event a is* **enabled** *at a global program state σ if there exists a program state σ' s.t. $(\sigma, a, \sigma') \in R$.*

In order to synchronize concurrent operations in a multithreaded program, the programming language provides synchronization operations that allow the programmer to constrain the order of operations among the different threads. In languages such as Java, these operations operate on runtime locations. It is also possible to support lower level synchronization operations such as Test&Set.

Definition 2.2.7 *A* **protected access event** *is an access event to a memory location that takes part in a synchronization operation.*

The `Lock(`$lock^x$`)` and `Unlock(`$lock^x$`)` operations that we added to Figure 2.2, in order to prevent the datarace, are synchronization operations. Therefore, `Lock(`$lock^x$`)` and `Unlock(`$lock^x$`)` are protected access events to a memory location "$lock^x$".

Definition 2.2.8 A datarace *in a multithreaded program occurs if there exists a reachable global state σ and two access events, a_1 and a_2, by different threads, such that the following conditions are met:*

1. *a_1 and a_2 access the same memory location m.*
2. *a_1 or a_2 are writing into m.*
3. *a_1 or a_2 are not protected access events.*
4. *a_1 and a_2 are enabled at σ.*

The intuition behind Definition 2.2.8 is that conditions 3 and 4 imply the existence of at least one prefix of a program trace π, such that $\pi.a_1.\sigma_1.a_2$

and $\pi.a_2.\sigma_2.a_1$ are valid prefixes of program traces. σ_1 and σ_2 are global program states. Figure 2.1 illustrates this intuition. Moreover, this example shows why dataraces are program bugs. On the other hand, the program fragment in Figure 2.2 does not contain a datarace because `Lock(`$lock^x$`)` and `Unlock(`$lock^x$`)` enforce an order between the accesses to `X`. Moreover, there is no datarace on "$lock^x$" because all accesses to "$lock^x$" are protected access events, in contrast to condition 3 above.

2.3 The Lockset algorithm

In this section, we describe a simple version of the Lockset algorithm. The algorithm monitors all access events and lock acquisitions. It is based on the assumption that well-behaved programs preserve a locking discipline, i.e., for every shared memory location, there exists a lock such that every access to the location is guarded by this lock. Thus, the algorithm guarantees the absence of dataraces in a given execution by checking that, for every shared memory location, there exists such a lock.

Furthermore, in many cases, violations of the locking discipline indicate dataraces in different thread interleavings caused by scheduling. This helps Lockset to predict dataraces in future executions and not just find errors in the current execution. However, this feature is most beneficial due to the exponential number of interleavings. The tradeoff is that Lockset cannot provide a witness for the datarace. In Section 2.4, we overcome this problem by means of a hybrid algorithm that constructs a witness for a datarace, even when it occurs only in rare interleavings.

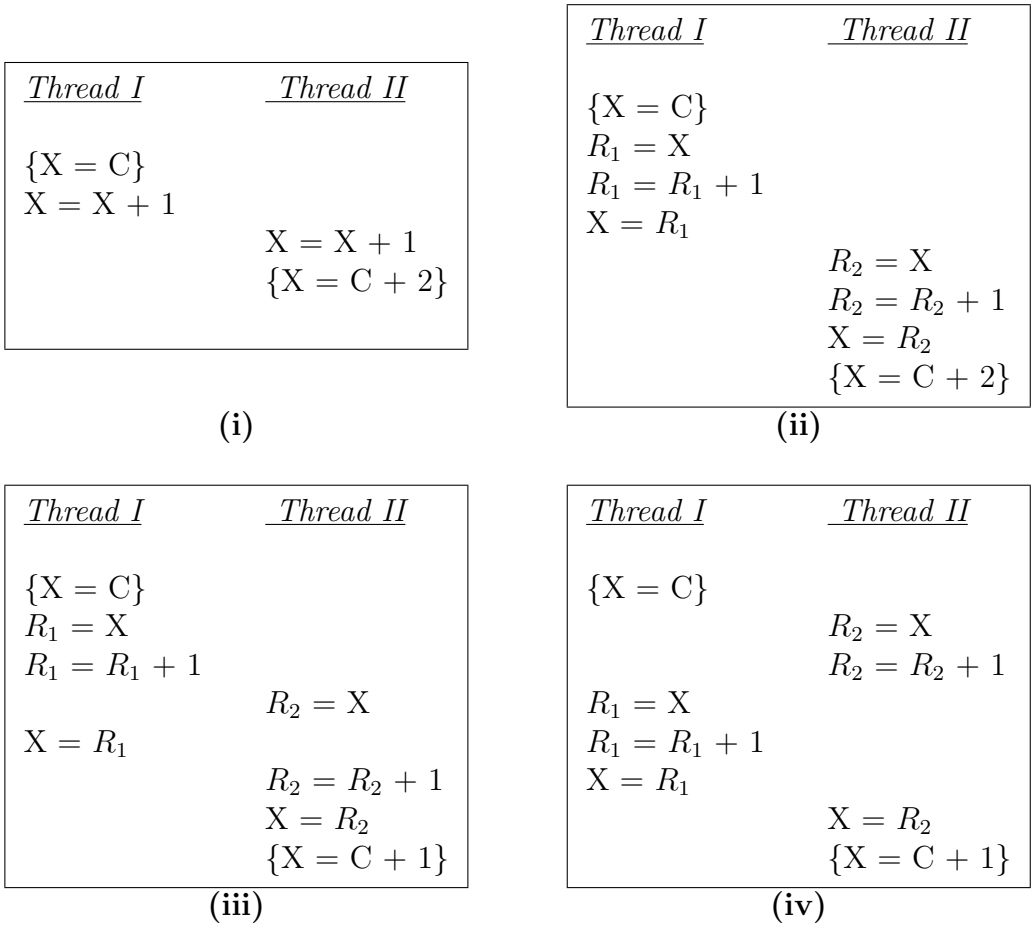
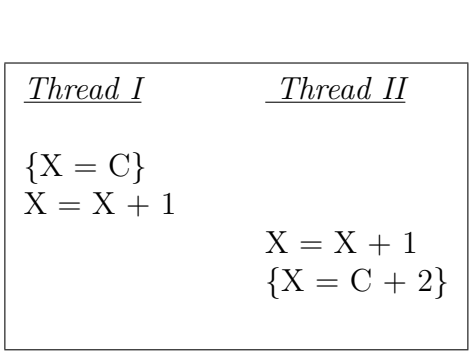
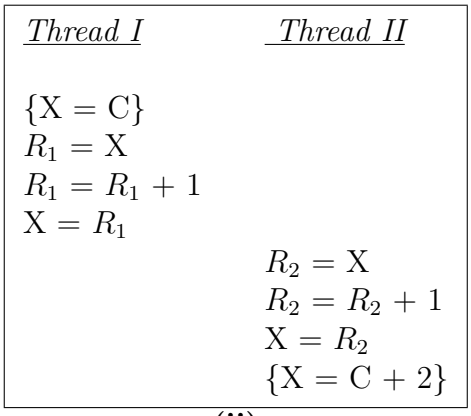


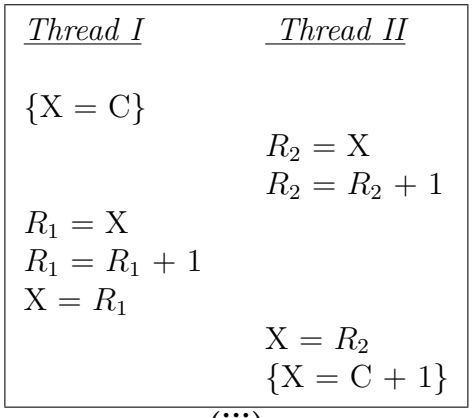
Figure 2.1: A datarace intuition example. (i) contains a program fragment with two threads such that each thread increments the value of a global variable `X` by 1. (ii) contains the same program as (i), but here we use two registers, R1 and R2, to split X =X + 1 into three atomic operations. (iii) and (iv) contain the same program, but with a different thread interleaving. It is easy to see that the output of this execution is `X = C + 1` and not the expected result `X = C + 2`.



(i)



(ii)



(iii)

Figure 2.2: An example without a datarace. (i) contains a program fragment with two threads such that each thread increments the value of a global variable `X` by 1. (ii) contains the same program as (i), but here we use two registers, R1 and R2, to split X =X + 1 into three atomic operations. (iii) contains the same program as (ii), but with a different thread interleaving.

Initialization	*Monitor*
for each shared memory m $C(m) = \Omega$	On access event a: $C(m^a) = C(m^a) \cap locks^a$ if $C(m^a) = \emptyset$ then display a warning

Figure 2.3: The Lockset algorithm. m^a denotes the memory location accessed by a. $locks^a$ are the locks that the thread performs a holds. Ω is the set that contains all the locks in the program.

Lockset checks whether a program adheres to the locking discipline by monitoring all reads and writes as the program executes. Since Lockset , before monitoring a program execution, has no way of knowing which locks are intended to protect which memory locations, it must infer this information from the execution history. For each shared memory location m, Lockset maintains the set $C(m)$ of candidate locks for m. This set contains those locks that have protected m for the computation so far. That is, a lock l is in $C(m)$ if, in the computation up to that point, every thread that has accessed m was holding l at the moment of the access.

When a new memory location m is initialized, its candidate set $C(m)$ is considered to hold all possible locks. When a memory location m is accessed, Lockset updates $C(m)$ with the intersection of $C(m)$ and the set of locks held by the current thread. This process, called Lockset refinement, ensures that any lock that consistently protects m is contained in $C(m)$. If some lock l consistently protects m, it will remain in $C(m)$ while $C(m)$ is refined. If $C(m)$ becomes empty, this indicates that no lock consistently protects m.

Figure 2.3 displays the pseudocode of Lockset.

Throughout this paper, we assume that the following information on every monitored access event a is available:

- The program counter of each thread.
- m^a, the shared memory location accessed.
- t^a, the thread that performs a.
- τ^a, the type of access a (Read or Write).
- ψ^a, whether access a is protected (True or False).
- $locks^a$, the locks that t^a holds when a occurs.
- The global program state, which includes the values of local and global variables as well as the content of the heap.

2.4 Running example

The applicability of our technique is illustrated by Figure 2.4, which gives a running example of a program that operates on shared data. Every so often, the threads of this program are synchronized in order to perform deletion operations on shared data. This program simulates a realistic situation in which concurrent programs share common data that needs to be reorganized once in a while. In addition, there are many concurrent programs which do not contain common data but whose threads are synchronized once in a while in order to share work. Our example illustrates a synchronization point. All the program threads execute the same code.

```
// Huge program fragment
⋮
1   synchronized(KeyLock) {
2           if (numDelItr%5 == 0)
3                   DB.Compress();
4   }

5   for (int i=myStart; i < myStart + (poolSize / numThreads); ++i) {
6           synchronized(NumLock) {
7                   if (DB.getClause(i).IsTooBig()) {
8                           DB.getClause(i).DeleteClause();
9                           ++NumDeleted;
10                  }
11          }
12  }

13  synchronized(DelLock) {
14          ++numDelItr; // Lockset reports a warning!
15  }
⋮
// Huge program fragment
```

Figure 2.4: A running example

The datarace in our example might occur on variable `numDelItr` between lines 2 and 14. This datarace occurs only on rare interleavings when a thread t_1 reaches line 14 while another thread t_2 is executing line 2.

The content can be large enough so model checkers cannot explore the huge code fragment before and after the relevant code. This is because the amount of data and the number of thread interleavings on this program skeleton is very large.

Figure 2.5 illustrates how Lockset identifies a violation of the locking discipline while monitoring our running example. For each $i \in \{1, 2\}$, column t_i shows the operations of thread t_i and the locks that t_i holds in each operation. The right column displays the candidate lock set of `numDelItr` during each operation. The rows of the figure illustrate the interleaving between t_1 and t_2. $C($`numDelItr`$)$ is initialized to contain all the locks of the program, Ω. After `numDelItr` is accessed by t_1 and while t_1 holds `KeyLock`, $C($`numDelItr`$)$ is refined to contain that lock. `numDelItr` is accessed again by t_1 while t_1 holds `DelLock`, and then $C($`numDelItr`$)$ is refined to the intersection of $\{$`KeyLock`$\}$ and $\{$`DelLock`$\}$, which is the empty set ϕ. The empty set indicates that there is no consistent lock protecting `numDelItr`.

In this section, we show how to utilize a model checker in order to locate the actual thread interleaving in which a datarace occurs. Recall that model checking is size sensitive, and therefore, employing it alone for realistic programs usually leads to state space explosion. On the other hand, employing Lockset alone for datarace detection results in many spurious dataraces and no trace for each warning. Hence, we use Lockset information in order to help the model checker locate the actual thread interleaving in which a datarace occurs. In general, locating such interleavings is very challenging, because the actual number of potential interleavings in realistic programs is large. Fortunately, we can use the information generated by Lockset to make this task feasible. The main idea is to restrict the set of potential interleavings investigated by the model checker according to the prefix of the actual runtime trace executed by the Lockset algorithm. This allows us to reduce the number of interleavings and, in particular, to reduce the number of threads which

Thread I		*Thread II*		C(numDelItr)
operations	*hold_lock*	*operations*	*hold_lock*	
⋮	ψ			Ω
		⋮	χ	Ω
synchronized(KeyLock)	{KeyLock}			Ω
if(numDelItr%5==0)	{KeyLock}			{KeyLock}
		synchronized(KeyLock)	{KeyLock}	{KeyLock}
		if(numDelItr%5==0)	{KeyLock}	{KeyLock}
⋮	ξ			{KeyLock}
		⋮	η	{KeyLock}
synchronized(DelLock)	{DelLock}			{KeyLock}
++numDelItr	{DelLock}			ϕ

Figure 2.5: An execution of Lockset on the running example shown in Fig. 2.4. The refinement operations are displayed in the rightmost column. A warning for a violation of the locking discipline is produced on the last line when $C(numDelItr)$ becomes empty.

need to be explored. Our algorithm chooses the particular prefix by exploiting locking information along the trace. This guarantees that our algorithm is compatible with state-of-the-art dynamic datarace detection tools, and can be extended in order to support new dynamic lock-based techniques. Moreover, the experimental results show that our approach can actually predict dataraces even on traces in which other tools are unable to locate them. In fact, our tool locates dataraces which occur on rarely executed interleavings, as is shown in Figure 2.4.

Our algorithm operates in two phases: In phase 1, it computes a witness prefix using Lockset information, and in phase 2 it generates a transition system and use a model checker that investigates different interleavings to find a witness suffix.

Definition 2.4.1 *A* **witness** *for a datarace is a prefix of a program trace (Π) with an access event a_1 by a thread t_1 and an access event a_2 by a different thread t_2 to the same memory location m, such that the following conditions are met:*

1. *a_1 or a_2 are writing into m.*
2. *a_2 is the first action after a_1 on Π.*
3. *a_1 or a_2 are not protected access events.*

Recall that our witness definition reveals a datarace by virtue of the fact that a_1 can actually be postponed until after a_2 is executed. It is easy to see that a witness for a datarace between accesses a_1 by t_1 and a_2 by t_2 exists

in a multithreaded program if and only if there exists a datarace between accesses a_1 by t_1 and a_2 by t_2 in the same program.

In each locking discipline violation warning, Lockset provides only a single access event a. The first phase of our algorithm constructs an extra access event a_1 that can take part in a race with a, and the second phase uses a model checker to construct the trace which satisfies the above requirements for $a_2 = a$. a_1 determines the prefix of witnesses explored by our algorithm.

2.5 Phase 1: Finding a prefix for a witness

This section provides an algorithm for finding an access event a_1 before the violation of the locking discipline. The algorithm is shown in Figure 2.6. This algorithm performs a backward scan on the access events gathered by Lockset, starting from the violation location (a_2). The algorithm locates the last access event in the execution (before the violation) which satisfies the following conditions:

- $m^{a_1} = m^{a_2}$
- $t^{a_1} \neq t^{a_2}$
- $(\tau^{a_1} = Write)$ or $(\tau^{a_2} = Write)$
- $(\psi^{a_1} = False)$ or $(\psi^{a_2} = False)$
- $locks^{a_1} \cap locks^{a_2} = \emptyset$

The first four conditions reflect the definition of a datarace. The last condition naturally reflects the locking discipline, i.e., a_1 and a_2 do not have

a mutual lock which protects m^{a_2}. This algorithm terminates abnormally when a second such access event a_1 cannot be found. In some cases this indicates a spurious Lockset warning, as in Figure 2.7. But it may also indicate a limitation of our current approach, as in Figure 2.8. However, if the algorithm for finding a_1 succeeds in providing a witness prefix up to a_1, it is possible to drastically reduce the cost of the second phase by providing a larger prefix. This phase heuristically reduces the cost of model checking by delaying σ_{a_1} as much as possible. For that reason, we scan for the last event a_1. σ_{a_1} is a global program state that appears right before a_1 on the witness prefix provided by the algorithm for finding a_1.

Figure 2.9 displays the pseudocode of the algorithm. The main idea is to set the initial configuration explored by the model checker to exclude operations which cannot be affected by a_1 or by operations of t^{a_1} after a_1. The algorithm conservatively excludes these operations in time proportional to the execution trace between σ_{a_1} and a_2. This algorithm can provide a witness for a datarace if, during the traversal, an access event that can take part in a race with a_1 is found.

Figure 2.10 shows the prefix that the algorithm chooses for the execution of the running example that was shown in Figure 2.5.

```
FindRacingEvent(access_event_stack)
        while(¬ empty(access_event_stack))
        {
              set a1 = pop(access_event_stack)
              if(m^a1 = m^a2 ∧ t^a1 ≠ t^a2
                 ∧ (τ^a1 = Write ∨ τ^a2 = Write)
                 ∧ (ψ^a1 = False ∨ ψ^a2 = False)
                 ∧ (locks^a1 ∩ locks^a2 = φ))
                              return a1 //found a racing access event
        }
        exit //there is no access event which satisfies the requirements.
```

Figure 2.6: An algorithm for finding an access event a_1 which can take part in a race on a memory location m^{a_2}. In order to locate the last access event we use a stack *access_event_stack* which contains all the access events information that was gathered by Lockset during the runtime execution.

Thread I		*Thread II*		*C(X)*
operations	*hold_lock*	*operations*	*hold_lock*	
Lock($lock^x$)	$\{lock^x\}$			$\{lock^x, lock^y\}$
Lock($lock^y$)	$\{lock^x, lock^y\}$			
X = Y				
Unlock($lock^y$)	$\{lock^x\}$			
Unlock($lock^x$)	ϕ			
		Lock($lock^x$)	$\{lock^x\}$	
		X = 7		$\{lock^x\}$
		Unlock($lock^x$)	ϕ	
		Lock($lock^y$)	$\{lock^y\}$	
		Y = X		ϕ
		Unlock($lock^y$)	ϕ	

Figure 2.7: An example without a datarace. In this example, the algorithm in Figure 2.6 terminates abnormally while searching for a_1. The reason is that Lockset displays a violation warning on `X` when `C(X)` becomes ϕ, and there is no previous access event in this execution which satisfies the conditions from 2.5. In addition, there is no possibility of a datarace on `X` in this code fragment, because synchronization operations prevent `X = Y` from occurring concurrently with `X = 7` or `Y = X`.

Thread I		*Thread II*		*C(X)*
operations	*hold_lock*	*operations*	*hold_lock*	
		Lock($lock^x$)	$\{lock^x\}$	$\{lock^x, lock^y\}$
		Lock($lock^y$)	$\{lock^x, lock^y\}$	
		Y = X		
		Unlock($lock^y$)	$\{lock^x\}$	
		Unlock($lock^x$)	ϕ	
Lock($lock^x$)	$\{lock^x\}$			
Lock($lock^y$)	$\{lock^x, lock^y\}$			
Y = X				
Unlock($lock^y$)	$\{lock^x\}$			
Unlock($lock^x$)	ϕ			
		Lock($lock^x$)	$\{lock^x\}$	
		X = 7		$\{lock^x\}$
		Unlock($lock^x$)	ϕ	
		Lock($lock^y$)	$\{lock^y\}$	
		Y = X		ϕ
		Unlock($lock^y$)	ϕ	
Lock($lock^x$)	$\{lock^x\}$			
X = 7				
Unlock($lock^x$)	ϕ			

Figure 2.8: An example with a datarace not captured by our technique. The datarace occurs between the bolded operations. The reason is that there is no previous access event, to the warning location, which can take part in a race with it. This problem can be solved by going forward on the access event list, starting from the warning location.

```
1  ExtendConstructWitnessPrefix(Π, π, V)
2      Let Π be the runtime execution prefix until a_1 (exclusive)
3      Let π be the trace from a_1 until a_2
4      Let V be the bit vector for the global memory locations
5      Let ψ be the set of the stuck threads, initialized by t^{a_1}
6      while (exists another operation of π)
7          set α = next operation on π
8          if (α is performed by a thread t s.t t ∈ ψ)
9              if (α is a Write operation to a global memory location m)
10                 set V[m] = 1
11             continue to the next operation
12         if (m^α ≠ m^{a_1} ∧ (τ^α = Write ∨ τ^{a_1} = Write) ∧ (¬ψ^α ∨ ¬ψ^{a_1}))
13             add α to Π
14             return Π // a datarace was found!!!
15         if (α performs on a local memory location)
16             add α to Π
17         if (α performs a Write operation to a global memory location m)
18             add α to Π
19             set V[m] = 0
20         if (α performs a Read operation from a global memory location m)
21             if (V[m] = 0)
22                 add α to Π
23             else
24                 add the thread that perform α to ψ
25                 if (|ψ| = number of threads)
26                   return Π
27         if (α tries to acquire a lock ℓ)
28             if (ℓ is owned by a thread t s.t t ∈ ψ)
29                 add the thread that perform α to ψ
30                 if (|ψ| = number of threads)
31                   return Π
32             else
33                 add α to Π
34         if (α tries to release a lock)
35             add α to Π
```

Figure 2.9: An algorithm that constructs a prefix of a witness using the runtime information. V is a bit vector that is initialized to zero. Each memory location m is mapped to a unique entry in $V[m]$. ψ is a set that contains all the threads that got stuck during the analysis.

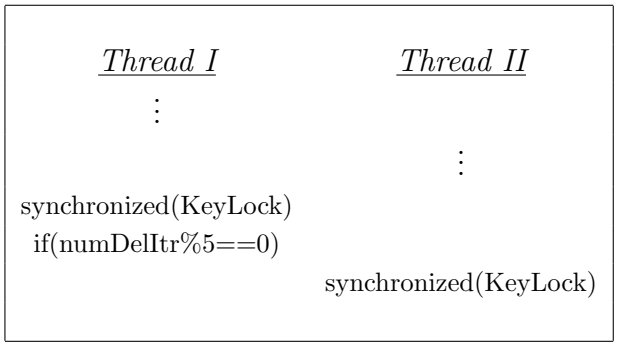


Figure 2.10: The prefix that the algorithm from Figure 2.6 chooses from the execution of our running example in Figure 2.5.

2.6 Phase 2: Constructing witnesses using a model checker

In this section, we generate a witness for a datarace on a memory location m^{a_2} using model checking techniques.

2.6.1 Constructing a model

We construct a model for the program fragment between σ_{a_1} and a_2. An alternative method is to first construct a model for the whole program and then reduce the size of the model by eliminating parts which do not occur between σ_{a_1} and a_2. Recall that the access events a_1 and a_2 include the program counters and the thread identification.

How to build a model from a program is explained in [2]. The pseudocode for constructing the model is shown in Figure 2.11. It consists of three phases:

- Building a model M of the program. We add the following: another

```
BuildModel(Π)
    Let Π be the prefix
    build program model M without t^{a_1}
    for each thread t in M
        set t's local variable values to their last value on Π
        set t's program counter to its last value on Π
    for each global variable gv
        set gv's value as its last value on Π
    perform chopping
```

Figure 2.11: An algorithm for building a model of the program fragment for all possible witness suffixes.

sink state, $\sigma_{overflow}$, to M, tuples (σ, ac, $\sigma_{overflow}$) to R for every state σ, and an action ac that causes variable overflow. In addition, we add tuples ($\sigma_{overflow}$, ac, $\sigma_{overflow}$) for each action ac.

- For each thread t which takes part in the new model, setting the initial values of all the local variables of t to their value at σ_{a_1}, and, in addition, setting the initial value of the program counter of t to its last value at σ_{a_1}.

- Setting the initial value of each global variable to its value at σ_{a_1}.

2.6.2 Further reduction of the model size

After the model has been built, we use a model checker to detect dataraces by exploring all the thread interleavings of the program (see next section).

This technique is powerful and provides very good results. However, in order to further increase the power of our technique, we suggest a heuristic which excludes a thread from the model. This heuristic reduces drastically the size and complexity of the model and helps the model checker find a witness for a datarace. Definition 2.4.1 shows that a witness for a datarace should include two access events, a_1 and a_2, by different threads. Therefore, by setting the initial state of the model to be the global program state σ_{a_1}, we guarantee that t^{a_1} will perform a_1. Hence, even when t^{a_1} is excluded, finding a path in the model in which a_2 is performed by t^{a_2} provides enough information for building a witness.

Finally, it is important to note that the cost of model checking is reduced by:

- The reduction in model size – Our work focuses on the program fragment between a_1 and a_2. Therefore, we can employ strong reductions, such as program chopping [24], which reduce the size of the model drastically.

- The elimination of thread t^{a_1} – the elimination of t^{a_1} from the model removes t^{a_1}'s transition, t^{a_1}'s local variables, and all the interleavings with t^{a_1}.

- Providing a single new initial configuration σ_{a_1} – providing a deterministic initial state.

- Heuristically reducing the number of steps that the model checker should carry out in order to find a datarace.

2.6.3 Using a model checker

Finally, we employ a model checker to check whether there exists a witness suffix. Specifically, we run the model checker on the model M (see Figure 2.11) with a property requiring that a_1 and a_2 can be performed one after the other. Recall that the memory location and thread ID is part of the access event. The model checker does an exhaustive search and checks all the thread interleavings in M in order to determine whether there exists a thread interleaving such that a_1 and a_2 are performed one after the other. If we use the heuristic from the previous section, then the property should require that t^{a_2} perform a_2.

Figure 2.12 displays a valid program trace that our technique, up to this stage, generates for a datarace on variable `numDelItr` in our running example from Figure 2.4.

Throughout this thesis, we denote the infinite model of the multithreaded program by MP.

Lemma 2.6.1 *Every program trace π^M in M such that $\sigma_{overflow} \notin \pi^M$ is a trace in MP.*

Proof

M is built in several phases:

1. We build a finite model of the program by using an under-approximation and only represent a constant number of bits per integer. This constant should be large enough to represent σ_{a_1}. We add the following: another sink state$\sigma_{overflow}$ to M, tuples (σ, ac, $\sigma_{overflow}$) to R for every state σ,

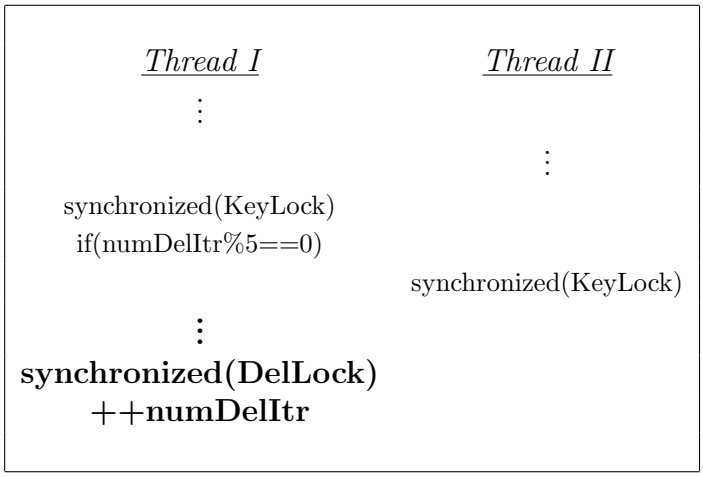


Figure 2.12: A valid program trace that our technique, up to this stage, generates for a datarace on variable `numDelItr` in our running example from Figure 2.5. The suffix found by the model checker is highlighted.

and an action ac that causes a variable overflow. In addition, we add tuples ($\sigma_{overflow}$, ac, $\sigma_{overflow}$) for each action ac.

2. We change the set of initial states of M to be the global program state at a_1 (σ_{a_1}).

3. We perform conservative reduction, such as program chopping, on the model.

In this proof, we denote the model after phase 2 by M', and the relation of a model $\bar{M}$ by $R^{\bar{M}}$. In addition, we denote the set of initial states of a model $\bar{M}$ by $\Sigma_0^{\bar{M}}$ and the set of states of $\bar{M}$ by $\Sigma^{\bar{M}}$.

M' represents a finite model of MP with a constant number of bits per

integer and an additional sink state $\sigma_{overflow}$. Therefore, $R^{M'} \mid_{\Sigma^{M'} \setminus \sigma_{overflow}} = \{(\sigma, ac, \sigma') \mid (\sigma, ac, \sigma') \in R^{M'} \wedge \sigma' \neq \sigma_{overflow}\} \subseteq R^{MP}$, which means that every tuple $(\sigma_i, ac_i, \sigma_{i+1})$ in $R^{M'} \mid_{\Sigma^{M'} \setminus \sigma_{overflow}}$ is in R^{MP}. In addition, $\Sigma_0^{M'} = \{\sigma_{a_1}\}$ and we know, using Lockset information, that σ_{a_1} is reachable in MP. Let $\pi = \sigma_{a_1}, ac_1, \sigma_2, ac_2, \ldots$ be a program trace in M' such that $\sigma_{overflow} \notin \pi$. From Definition 2.2.3 we know that each i such that $\sigma_i, ac_i, \sigma_{i+1} \in \pi$ implies that $(\sigma_i, ac_i, \sigma_{i+1}) \in R^{M'}$. $\sigma_{overflow}$ is not in π, and we know that $R^{M'} \mid_{\Sigma^{M'} \setminus \sigma_{overflow}} \subseteq R^{MP}$ and that σ_{a_1} is reachable at MP. Hence, each i such that $\sigma_i, ac_i, \sigma_{i+1} \in \pi$ implies that $(\sigma_i, ac_i, \sigma_{i+1}) \in R^{MP}$. We conclude that for every program trace π, if $\pi \in M'$ then $\pi \in M^{MP}$.

M has the same initial state as M' but, because of the chopping reduction, $R^M \subseteq R^{M'}$. Therefore, for every program trace π^M, if $\pi^M \in M$ then $\pi^M \in M'$ and, as we have already shown, π^M is a trace in MP. ∎

Lemma 2.6.2 *Let π_1 be a program trace prefix provided by Lockset, up to a_1 (exclusive), and let π^{MP} be a trace in MP such that the first state in π^{MP} is σ_{a_1}. Then the concatenation of π_1 and π^{MP} is a valid program trace.*

Proof

Let π_1 be a program trace prefix provided by Lockset. π_1 is clearly a valid program trace prefix because Lockset generates π_1 while executing the program. Moreover, using Lockset information, we know that there exists a program trace prefix π', a global program state σ', and an action ac' such that $\pi_1 = \pi'.\sigma'.ac'$ and $(\sigma', ac', \sigma_{a_1}) \in R$. Let $\pi = \sigma_0, ac_0, \sigma_1, ac_1, \ldots$ be a trace from the program such that $\sigma_0 = \sigma_{a_1}$. Because $(\sigma', ac', \sigma_{a_1}) \in R$, we conclude that $\pi'.\sigma'.ac'.\sigma_{a_1}.ac_0.\sigma_1.ac_1 \ldots$ is a valid program trace, and hence,

$\pi_1.\pi$ is a valid program trace as well.

■

Lemma 2.6.3 *If a trace π_2 is returned by the model checker, then $\sigma_{overflow} \notin \pi_2$.*

Proof

If the model checker returns a trace π_2, then $\pi_2 = \sigma_{a_1}.ac_1 \ldots a_2$. Because $a_2 \neq \sigma_{overflow}$ and $\{\sigma_{overflow}\} = \{\sigma \mid \exists ac \ s.t. \ (\sigma_{overflow}, ac, \sigma) \in R\}$, we conclude that $\sigma_{overflow} \notin \pi_2$.

■

Corollary 2.6.1 *Lemmas 2.6.1, 2.6.2 and 2.6.3 imply that the concatenation of a program trace prefix π_1 provided by Lockset and a program trace π_2 ($\pi_2 \in M$) provided by the model checker ($\pi_1.\pi_2$) is a valid program trace prefix.*

After showing that $\pi_1.\pi_2$ is a valid program trace, we are motivated to show how to convert this program trace to a witness for a datarace. If the heuristic which excludes t^{a_1} from the model is not used, then the concatenation of π_1 and π_2 is already a witness. Otherwise, there is an additional step for completing the witness. The model checker checks for reachability of a_2 by t^{a_2}. Hence, there exists a program trace prefix π and a global program state σ_{a_2} such that $\pi_1.\pi_2 = \pi.\sigma_{a_2}.a_2$. In order to complete the witness, we need to calculate a new state $\sigma_{post\ a_1}$ such that $(\sigma_{a_2},\ a_1,\ \sigma_{post\ a_1}) \in R$, and add $\sigma_{post\ a_1}$ and a_1 to $\pi_1.\pi_2$ to generate a trace $\pi.\sigma_{a_2}.a_1.\sigma_{post\ a_1}.a_2$.

Lemma 2.6.4 guarantees that the generated trace is a valid witness for a datarace. Thus, our method produces no spurious alarms and creates witnesses only for real dataraces.

Lemma 2.6.4 *A trace ω generated by our technique is a witness for a datarace on m^{a_1} between t^{a_1} and t^{a_2}.*

Proof

Recall that a witness for a datarace (Definition 2.4.1) is a prefix of a program trace (Π) with an access event a_1 by a thread t_1 and an access event a_2 by a different thread t_2 to the same memory location m, such that the following conditions are met:

1. a_1 or a_2 are writing into m.
2. a_2 is the first action after a_1 on Π.
3. a_1 or a_2 are not protected access events.

The algorithm for finding an access event a_1 (Section 2.5) guarantees that:

- $m^{a_1} = m^{a_2}$
- $t^{a_1} \neq t^{a_2}$
- $(\tau^{a_1} = Write)$ or $(\tau^{a_2} = Write)$
- $(\psi^{a_1} = False)$ or $(\psi^{a_2} = False)$

Therefore, conditions 1 and 3 are trivially satisfied. Consequently, we only need to show condition 2 and show that ω is a program trace prefix. Assume

that the heuristic for reducing a thread was not used in generating ω, and hence, $\omega = \pi_1.\pi_2$. Corollary 2.6.1 proves this lemma. Assume that the heuristic for reducing a thread was used in generating ω. Hence, there exists a program trace prefix π such that $\omega = \pi.\sigma_{a_2}.a_1.\sigma_{post\ a_1}.a_2$. Because $\pi.\sigma_{a_2}$ is a prefix of $\pi_1.\pi_2$ we conclude that π is a program trace prefix. t^{a_1} is not part of the model, which guarantees that there are no operations of t^{a_1} in π. Using the fact that a_1 is enabled at σ_{a_1} we conclude that a_1 is enabled at σ_{a_2}, which proves that $\pi.\sigma_{a_2}.a_1$ is a program trace prefix. In order to build ω, we calculate a global program state $\sigma_{post\ a_1}$ such that $(\sigma_{a_2}, a_1, \sigma_{post\ a_1}) \in R$. $\pi.\sigma_{a_2}.a_1.\sigma_{post\ a_1}$ is a program trace prefix. Finally, because a_2 is enabled at σ_{a_2} and $t^{a_1} \neq t^{a_2}$, we conclude that $\pi.\sigma_{a_2}.a_1.\sigma_{post\ a_1}.a_2$ is a program trace prefix in which a_2 is the first action after a_1. Therefore, ω is a witness for a datarace. ■

Figure 2.13 illustrates the witness that our technique generates for a datarace on variable `numDelItr` in the running example in Figure 2.5.

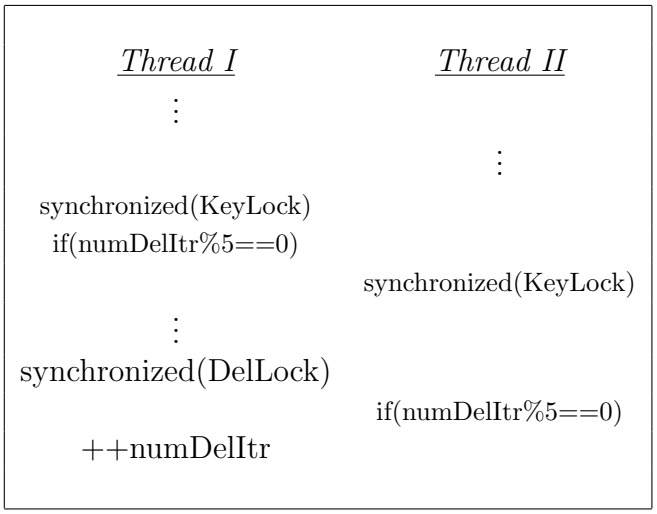


Figure 2.13: A witness that our technique generates for a datarace on variable `numDelItr` in the running example in Figure 2.5

.

Chapter 3

Prototype implementation

We have implemented a semi-automatic system, based on IBM tools, using the algorithm described in the previous section. The manual steps in our system can be fully automated. We have applied our prototype to some public-domain programs and to other programs that were enhanced to demonstrate the utility of our technique.

3.1 System description

Our tool operates in the following stages:

Stage I Launches the Lockset dynamic tool on a given multithreaded program in order to generate warnings for violations of the locking discipline (see Section 2). The IBM Watson tool from [7] was used.

The remaining stages are executed per violation warning.

Stage II Finds an additional access event a_1 as described in Section 2.5.

Stage III Reduces the size and depth of the model by generating a longer witness prefix. This stage is described in Section 2.5.

Stage IV Generates a transition system (model) for the model checker (see Section 2.6).

Stage V Applies the model checker in order to generate a trace for the transition system. We used the IBM Haifa tool Wolf, which is built on top of RuleBase Parallel Edition [18]. Wolf is a symbolic model checker tuned for software, which uses disjunctive partitioning [2]. We use an under-approximation and only represent a constant number of bits per integer.

Stage VI If a trace from the model checker is returned, it is used to generate a witness, by concatenating the trace with the prefix constructed in Stage III, as described in Section 2.6.

3.2 Benchmark programs

We used the benchmark programs shown in Table 3.1. In `our_tsp` program, we added one synchronization operation to protect the shared data `TspSolver.MinTourLen`. We also added a global variable that counts the number of times a function is called. These enhancements are aimed at eliminating the dataraces in most executions, but this makes finding any remaining dataraces a more complicated task. Finally, we also created two programs, `ElevSim` and `DQueries`, with dataraces on rare interleavings. The `ElevSim` program contains many synchronization operations and a lot of non-

Program	Description	Lines
`tsp`	A traveling salesman program from ETH [32]	706
`our_tsp`	An enhanced version of `tsp`	708
`mtrt`	A multithreaded raytracer from specjvm98 [28]	3751
`hedc`	A Web crawler kernel from ETH [32]	29948
`SortArray`	A program which performs a parallel sort on an array from [27]. A datarace was added	362
`PrimeFinder`	A program which finds all the prime numbers in a given interval from [27]. A datarace was added	129
`Elevsim`	An elevator simulator	150
`DQueries`	A shared database simulator	166

Table 3.1: Our benchmark programs

determinism. *DQueries*, a more complicated program, also contains many synchronization operations and a lot of nondeterminism.

3.3 Empirical results

Our preliminary experimental results are very encouraging. We are able to generate witnesses for the violation warnings produced by Lockset. In certain cases, these witnesses can be generated using the optimization described in Figure 2.9, with either a very fast application of the model checker, or without using one at all. Without Lockset information, Wolf was unable to construct a trace for any of our examples within a week.

Our experimental results were obtained on an Intel Xeon dual CPU

2.4GHz, 2.5GB RAM platform running Linux.

Table 3.2 shows the number of bits that we use to represent an integer, in the transition systems of our benchmark programs.

Table 3.3 shows the runtime and the memory that the model checker consumed while trying to find a witness suffix. In these runs we exclude t^{a_1} from the model, as explained in Section 2.4. It is easy to see that the runtime and memory consumption increase drastically when the number of threads increases. For this reason, we use a hint [5] to direct the model checker's search toward a_2. The hint is very simple and derives from the Lockset execution. For each model checker execution, we simply use a hint that tries to advance t^{a_2} as much as possible. In cases where there exists, in the model, a path to a_2 which only includes operations of t^{a_2}, this hint has a very strong effect on both runtime and memory consumption.

Table 3.4 shows the runtime and memory that the model checker consumed while using a hint to find a witness suffix. Running our tool on `PrimeFinder`, with 3 threads, clearly demonstrates the effectiveness of this simple hint. The model checker's runtime decreases from 44695.7 seconds to 2645.57 seconds and memory consumption decreases from 697 MB to 143 MB. Another good example is `DQueries` using 4 threads, which finished after 585.97 seconds using a hint and time-outd after a week without using a hint.

Table 3.5 displays the runtime and memory consumption of our model checker when t_1 was not excluded from the model. Adding t_1 to the model increases drastically the runtime and memory consumption of the model checker. We added this table in order to show that we are able to find witnesses in the model with t^{a_1}, even though our method was able to find

Program	Bits per integer
our_tsp	5
SortArray	7
PrimeFinder	8
Elevsim	5
DQueries	4
hedc	6
tsp	5

Table 3.2: The number of bits that we use to represent an integer, in the transition systems of our benchmark programs.

Program	2 threads		3 threads		4 threads	
	time	memory	time	memory	time	memory
our_tsp	35069.9	353	—	mem out	—	mem out
SortArray	569.39	123	7019.38	607	—	mem out
PrimeFinder	888.74	116	44695.7	697	timeout	—
Elevsim	33.02	28	1393.56	151	55518.9	720
DQueries	140.13	60	10158.7	321	timeout	—
hedc	2.66	11	11.37	17	24.33	17
tsp	35243.2	337	—	mem out	—	mem out

Table 3.3: The runtime and memory consumption of the Model Checker Wolf in finding suffixes for datarace witnesses. The memory consumption is displayed in megabytes and the time in seconds. The term timeout denotes a period of more than a week.

Program	2 threads		3 threads		4 threads	
	time	memory	time	memory	time	memory
`our_tsp`	35069.9	353	—	mem out	—	mem out
`SortArray`	569.39	123	1334.93	396	—	mem out
`PrimeFinder`	888.74	116	2645.57	143	4547.18	168
`Elevsim`	33.02	28	67.92	33	147.91	48
`DQueries`	140.13	60	201.84	89	585.97	136
`hedc`	2.66	11	7.33	12	9	17
`tsp`	35243.2	337	—	mem out	—	mem out

Table 3.4: The runtime and memory consumption of the Model Checker Wolf in finding suffixes for datarace witnesses. The hint used in these examples biased the scheduler toward t^{a_2}. The memory consumption is displayed in megabytes and the time in seconds.

dataraces without this addition. This demonstrates the effectiveness of our heuristic.

Program	2 threads		3 threads		4 threads	
	time	memory	time	memory	time	memory
`our_tsp`	—	mem out	—	mem out	—	mem out
`SortArray`	1041.34	400	—	mem out	—	mem out
`PrimeFinder`	1871.46	200	5355.14	268	17260.6	295
`Elevsim`	36.36	31	124.79	49	408.81	73
`DQueries`	202.33	96	462.94	135	1843.63	202
`hedc`	7	12	9.52	16	122.81	21
`tsp`	—	mem out	—	mem out	—	mem out

Table 3.5: The runtime and memory consumption of the Model Checker Wolf in finding suffixes for datarace witnesses. The hint used in these examples biased the scheduler toward t^{a_2}. In these examples we did not exclude t^{a_1} from the model. The memory consumption is displayed in megabytes and the time in seconds.

Chapter 4

Related work

There are many excellent works on static and dynamic datarace detection.

4.1 Static tools

Static tools can be employed to guarantee the absence of dataraces in all interleavings and to identify potential dataraces. Flanagan and Freund extended Java's type-checker to detect dataraces [12]. This tool was expanded in [13] in order to handle large programs. However, it may produce many spurious alarms. Yahav has developed a static over-approximation tool which handles an unbounded number of objects and threads. This tool is fairly precise but only handles small programs [33]. Warlock [29] is an annotation-based static datarace detection system for ANSI C programs. Aiken and Guy developed a static datarace detection tool [1] in the context of SPMD programs. Since SPMD programs use barriers instead of locks, there is no need to track the lock's acquisitions. There are also generic model checking tools which can identify dataraces (e.g., [16]).

4.2 Dynamic tools

Dynamic detection tools that can handle large programs with precision have been developed. Some of them are based on Lamport's *happens-before* partial order relation. One such program is Djit [19], which was developed by Itzkovitch et al. and uses time stamps. Though precise, these tools sometimes produce spurious warnings. Furthermore, they only report errors in the current interleaving. This limits their usefulness because dataraces are hard to reproduce.

Another approach for dynamic datarace detection is based on a *locking discipline* in which each shared memory location must have a lock to guard it. This approach makes race detection more effective by displaying warnings for dataraces which can occur in a thread interleaving which is different than the one that was monitored. Savage et al. developed Eraser [26], which is a dynamic tool that tries to preserve a locking discipline using *lock sets* (see Section 2). The problem with Eraser is the large number of spurious warnings which it can produce during runtime execution. Praun and Gross [32] have improved Eraser by using escape analysis in order to monitor only escaped memory locations. In addition, they check for dataraces only at the object level. This approach increases the performance of Eraser, but monitoring at the object level causes too many spurious warnings to be displayed. Choi et al. [7] have developed a lock-based dynamic tool. They reduce the number of spurious alarms the lock-based tools produce by using points-to-static analysis and filtering out all the races which the static analysis did not discover. In addition they handle Java's `start` and

`join` operations and their tool reports dataraces between two access events which are not guarded by a lock. This tool provides, for each warning, two access events that might take part in a race. But the tool does not provide a witness for a datarace between these two access events. Moreover, it does not guarantee the existence of a datarace between these two access events. Tools which combine the happens-before and lock-based approaches were developed in order to reduce the number of spurious alarms and increase coverage. Choi and O'Callahan [22] have developed a hybrid tool from their Lockset-based tool [7] and a weaker version of a happens-before race detection tool.

Our hybrid tool differs from these in that we construct a witness for dataraces which happens-before based race detection tools can only discover on the executed trace. This work was inspired by [23], which improves the performance of Lockset and Djit. However, the work in [23] could not detect witnesses for traces which were not explored by the dynamic algorithm.

4.3 Combined static and dynamic tools

Havelund [15] has developed a hybrid tool which combines Lockset and model checking. Havelund's work tries to exploit warning information which Lockset provides in order to filter out threads from the model. The filtering is done by creating a model which contains all the threads that caused warnings during the Lockset execution. Then, for each thread t in the model, all the threads on which t depends are inserted. The dependency relation is created using a simple dependency analysis based on the runtime execution. Havelund's work and ours complement each other in datarace detection, but

use different techniques. In particular, we are able to reduce the complexity of model checking realistic-sized applications using Lockset information.

4.4 Scaling model checking

Many approaches for scaling model checking have been proposed. Yuan et al.[34] reduce the size of the transition system by simulating a prefix of the trace according to information provided by a user or randomly selected. A model checker is then executed to explore all traces starting with the selected prefix. This approach is similar to ours in that dynamic execution is used to trim the size of the transition system. In our approach, Lockset only exploits real executions, which can be a limitation for our tool unless a deterministic scheduler is used. Such limitations can be overcome by nondeterministic execution (e.g., [11]). A unique aspect of our approach is that the dynamic execution checks whether the locking discipline is obeyed. This has several consequences, positive and negative. On the positive side, model checking need not be applied at all in programs which obey the locking discipline. In programs which violate it, our tool exploits the dynamic information on the point of failure to generate a small transition system. This transition system, which starts after the first access event that takes part in the datarace, does not include the thread which performs this first access. This feature drastically reduces the size of the model. On the negative side, the overhead of dynamic checking in our approach is higher, and our tool may miss dataraces in programs which violate the locking discipline.

4.5 Conclusion

Many researchers have tried to deal with the difficult problem of datarace detection. Our hybrid approach combines model checking and dynamic datarace detection to detect dataraces in large programs without producing spurious warnings. This approach is very promising, and there are many possible extensions to it. Dependency analysis can be performed in order to decide which threads should be modeled. In addition, other techniques such as explicit (e.g. [17]) or bounded (e.g. CBMC [10]) model checking can be employed.

Bibliography

[1] A. Aiken and D. Gay. Barrier inference. In *Symp. on Princ. of Prog. Lang.*, pages 342–354, 1998.

[2] S. Barner and I. Rabinovitz. Efficient symbolic model checking of software using partial disjunctive partitioning. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 35–30, Oct. 2003.

[3] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Trans. on Electronic Computing*, EC-15(5):757–763, Oct. 1966.

[4] A. Biere, A. Cimatti, E. Clark, and Y. Zhu. Symbolic model checking without bdds. In *Tools and Alg. for the Const. and Analysis of Sys.*, pages 193–207, 1999.

[5] R. Bloem, K. Ravi, and F. Somenzi. Symbolic guided search for ctl model checking. In *Design Automation Conference*, pages 29–34. ACM Press, 2000.

[6] G. Brat, K. Havelund, S. Park, and W. Visser. Model checking programs. In *IEEE International Conference on Automated Software Engineering*, Sept. 2000.

[7] J. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridha-

ran. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Conf. on Prog. Lang. Design and Impl.*, 2002.

[8] J. Choi, A. Loginov, and V. Sarkar. Static datarace analysis for multi-threaded object-oriented programs. In *Conf. on Prog. Lang. Design and Impl.*, 2003.

[9] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT press, December 1999.

[10] CMU. Cbmc. http://www-2.cs.cmu.edu/~modelcheck/cbmc/.

[11] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded java programs. *Concurrency and Computation: Practice and Experience*, 15(3–5):485–499, 2003.

[12] C. Flanagan and S. Freund. Type-based race detection for java. In *Conf. on Prog. Lang. Design and Impl.*, 2000.

[13] C. Flanagan and S. Freund. Detecting race conditions in large programs. In *Work. on Prog. Analysis for Soft. Tools and Eng.*, pages 90–96. ACM Press, 2001.

[14] P. Godefroid. Verisoft: a tool for the automatic analysis of concurrent reactive software (short paper). In *Computer Aided Verification*, volume 1254 of *Lec. Notes in Comp. Sci.*, pages 476–479. Springer-Verlag, June 1997.

[15] K. Havelund. Using runtime analysis to guide model checking of java programs. In *SPIN*, pages 245–264, 2000.

[16] T. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *Computer Aided Verification*, pages 262–274, July 2003.

[17] G. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.

[18] IBM. Rulebase parallel edition. http://www.haifa.il.ibm.com/projects/verification/RB_Homepage /index.html, 2004.

[19] A. Itzkovitz, A. Schuster, and O. Zeev-Ben-Mordehai. Towards integration of data race detection in dsm systems. *Journal of Par. and Dist. Comp.*, 59(2):180–203, Nov. 1999.

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

[21] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, Sept. 1979.

[22] R. O'Callahan and J. Choi. Hybrid dynamic data race detection. In *Symp. on Princ. and Prac. of Par. Prog.*, pages 167–178. ACM Press, 2003.

[23] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded c++ programs. In *Symp. on Princ. and Prac. of Par. Prog.*, pages 179–190. ACM Press, 2003.

[24] T. Reps and G. Rosay. Precise interprocedural chopping. In *Symp. on Found. of Soft. Eng.*, pages 41–52. ACM Press, 1995.

[25] A. Roychoudhury and T. Mitra. Specifying multithreaded java semantics for program verification. In *International Conference on Software Engineering*, pages 489–499, 2002.

[26] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM*

Trans. on Comp. Sys., 15(4):391–411, 1997.

[27] K. Shagin. Benchmark programs. http://www.cs.technion.ac.il/~konst/benchmarks/, 2004.

[28] T. standard performance evaluation corporation. Spec jvm98 benchmarks., 1996. http://www.spec.org/osg/jvm98.

[29] N. Sterling. Warlock: a static data race analysis tool. In *USENIX Winter Technical Conference*, pages 97–106, 1993.

[30] S. D. Stoller. Model checking multi-threaded distributed java programs. In *SPIN*, pages 224–244, 2000.

[31] R. N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Inf.*, 19:57–84, 1983.

[32] C. von Praun and T. Gross. Object race detection. In *Conf. on OO Prog., Sys., Lang., and App.*, pages 70–82. ACM Press, 2001.

[33] E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *Symp. on Princ. of Prog. Lang.*, 2001.

[34] J. Yuan, J. Shen, J. A. Abraham, and A. Aziz. On combining formal and informal verification. In *Computer Aided Verification*, pages 376–387, 1997.