Available online at www.sciencedirect.com
ScienceDirect
J. Parallel Distrib. Comput. 67 (2007) 536–550
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

Scaling model checking of dataraces using dynamic information

Ohad Shacham[a,b,*,1], Mooly Sagiv[b], Assaf Schuster[c]

[a] *IBM Haifa Research Lab, Haifa, Israel*
[b] *School of Computer Science, Tel Aviv University, Israel*
[c] *Computer Science Department, Technion, Haifa, Israel*

Received 14 September 2005; received in revised form 11 October 2006; accepted 23 January 2007
Available online 9 February 2007

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 hybrid technique succeeds in providing thread interleavings that prove the existence of dataraces in realistic programs. The programs we experimented with cannot be checked using either an ordinary industrial strength model checker or bounded model checking.

Keywords: Datarace; Data race detection; Model checking; Lockset; Multithreading

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.

One of the key difficulties in detecting dataraces is that they may occur only on rare execution paths. In general, datarace detection for arbitrary programs is undecidable [4], and even for a restricted set of programs, such as those without branching, datarace detection is NP-complete [30]. Because dataraces are a cause for concern, many works have dealt with the problem of detecting them. Most works rely on static [9,14,33] or dynamic [20,23,26] analysis to find dataraces, while others use model checking [11], or a combination of techniques [8,16] for better results.

Static analysis tools such as [9,14,33] 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. Although dynamic analysis tools are more precise than their static counterparts, these tools can still produce spurious warnings. Furthermore, they report errors only for dataraces in the current interleaving, which limits their usefulness because dataraces are hard to reproduce. One such dynamic analysis tool is Djit [20], which is based on Lamport's happens-before partial-order relation and uses *time vectors* [21].

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

* Corresponding author. Fax: +972 4 829 6114.
E-mail addresses: ohads@il.ibm.com (O. Shacham), msagiv@post.tau.ac.il (M. Sagiv), assaf@cs.technion.ac.il (A. Schuster).
[1] This research was supported by The Israel Science Foundation (Grant No. 304/03).

0743-7315/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2007.01.006

discipline. The discipline requires that for every shared memory location there exists a lock such that all accesses to this location are 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 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 major 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 a 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 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.

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) at least one of a_1 or a_2 is a write operation;
(2) a_2 is the first action after a_1 on Π; and
(3) at least one of a_1 or a_2 is not a protected access event.

We define a protected access event as an access to a memory location that takes part in a synchronization operation. This synchronization operation can either be defined by the hardware, as in Test&Set, or by the programming language, as in Java `synchronized` operations. 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.

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 are capable of producing such witnesses. In addition, the improvement of formal verification tools will increase the usefulness of our method.

Symbolic model checking tools use a *transition system* to represent the possible computations of a program. Without using our hybrid technique, the transition systems of the programs that we checked were huge, in some cases beyond the capability of our software tuned model checker. We also tried to employ a bounded model checker [5], but still failed to find witnesses in our programs using these two techniques. Our hybrid method, however, 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 [18] or bounded [12] model checking.

We enhanced one example program by adding a synchronization operation in a way that creates a datarace that appears only when a rare interleaving is executed. This was done to demonstrate the usefulness of our technique. Because dataraces in our example programs occur on rarely executed paths, they are hard to find. Dynamic tools such as Djit (see Section 5.2) cannot find such dataraces when executing different traces. As expected, Lockset finds violations of the locking discipline even on traces that do not actually have dataraces, and the model checker produces witnesses on all our examples using Lockset information.

The rest of this paper is organized as follows: Section 2 formally defines dataraces and explains the Lockset algorithm. Section 3 describes our algorithm. A prototype implementation of the algorithm together with initial examples are described in Section 4. Related works are described in Section 5. We conclude in Section 6.

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 a 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* [22], 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. The sequential consistency assumption also means that, in every execution, program operations can be serialized.

Although many works for model checking of multithreaded programs [7,29] assume a sequentially consistent memory model, this assumption is problematic. Several memory models—Java, for example,—are in fact weaker. 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. Our approach can be extended to support their formal specification.

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. In this work we refer to a *global program state* as a tuple which contains: (i) a program counter value for each thread; (ii) values of all the global variables; (iii) values of local variables for each thread; (iv) the content of the heap.

We mark by Σ_0 the set of a program's initial states. We say that a tuple (σ, ac, σ'), where σ, σ' are global program states and ac is an action, is in a *relation* R ($(\sigma, ac, \sigma') \in R$) if the multithreaded program can step from σ to σ' by performing the action ac. A serialization of such global states connected by actions, such that every two consecutive states σ, σ' and their connecting action ac are in R, is called a *trace*. More formally, $\pi = [\sigma_0, ac_0, \sigma_1, ac_1, \sigma_2, \ldots,]$ is a trace if $\forall i.\sigma_i, ac_i, \sigma_{i+1} \in \pi$, $(\sigma_i, ac_i, \sigma_{i+1}) \in R$. A trace π is a *program trace* if the first state in π is in Σ_0. Every state on a program trace is reachable, hence, we say that a global program state σ is *a reachable program state* if there exists a program trace π such that $\sigma \in \pi$.

An *access event* is a read or a write operation to a memory location. An access event is a *protected access event* if it takes part in a synchronization operation. A synchronization operation can either be defined by the hardware, as in Test&Set, or by the programming language, as in Java `synchronized` operations. An access event a is *enabled* at a global program state σ if there exists a program state σ' s.t. $(\sigma, a, \sigma') \in R$.

Definition 2.1. A *datarace* in a multithreaded program occurs if there exists a reachable global state σ and two access events, a_1 and a_2, performed by different threads, such that the following conditions are met:

(1) a_1 and a_2 access the same memory location m;
(2) at least one of a_1 or a_2 is a write operation;
(3) at least one of a_1 or a_2 is not a protected access event; and
(4) a_1 and a_2 are enabled at σ.

The intuition behind Definition 2.1 is that conditions 2.1 and 2.1 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, where σ_1 and σ_2 are global program states. Such nondeterministic access to a single memory location from different threads may indicate a bug.

Fig. 1 illustrates this intuition, and shows how dataraces may indicate program bugs.

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 that states that for every shared memory location, there exists a lock that guards all accesses to this location. 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 because of the exponential number of interleavings. The tradeoff is that Lockset cannot provide a witness for the datarace. In Section 3, 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.

Lockset checks whether a program adheres to the locking discipline by monitoring all reads and writes as the program executes. Since Lockset has no way of knowing in advance which locks are intended to protect which memory locations, it must infer the protection relation 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 at all accesses in the computation so far. That is, a lock l is in $C(m)$ (at a certain point in time) if, during the computation up to this point, every thread that accessed m was holding l at the moment of 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. Fig. 2 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.

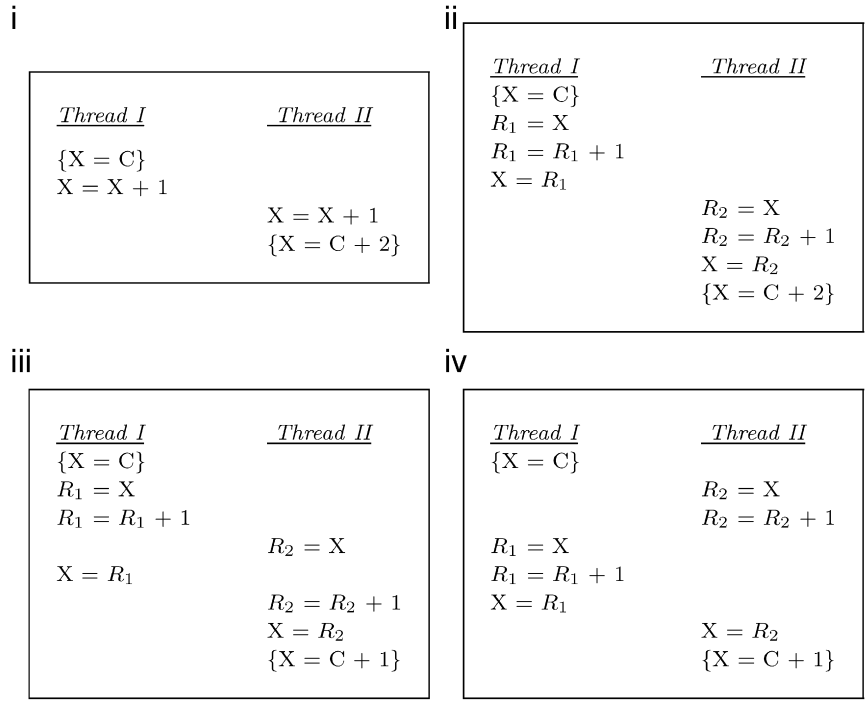

Fig. 1. A datarace intuition example: (i) contains a program fragment with two threads such that each thread increments the value of a global variable `X` by 1, (ii) contains the same program as (i), but here we use two registers, R1 and R2, to split $X = X + 1$ into three atomic operations, (iii) and (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`.

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

Fig. 2. The Lockset algorithm. m^a denotes the memory location accessed by a. $locks^a$ are the locks held by the thread that executes a. Ω is the set that contains all the locks in the program.

```
1  ModelChecking(M, φ)
2      Let M be a model.
3      Let φ be a specification.
4      Seen = Frontier = M.Σ0
5      While (Frontier ≠ ∅)
6          if (Frontier ∩ ¬φ ≠ ∅)
7              return "The specification does not hold (M ⊭ φ)"
8          Frontier = {σ' | ∃σ, ac : σ ∈ Frontier ∧ (σ, ac, σ') ∈ M.R}
9          Frontier = Frontier ∩ ¬Seen
10         Seen = Seen ∪ Frontier
11     return "The specification holds (M ⊨ φ)"
```

Fig. 3. Model Checking algorithm. $M.R$ denotes the relation of M. $M.\Sigma_0$ are the initial states of M. $M.\Sigma$ is the set of all states of M. *Seen* and *Frontier* are auxiliary sets.

- 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 is being executed.
- The global program state (σ_a), which includes the values of local and global variables as well as the content of the heap.

2.4. Model checking

In this section, we describe the notion of Model Checking [11]. Model Checking is a technique for verifying finite state machines. The model checker performs an exhaustive search on the state machine's state space in order to verify or falsify a given specification. Many Model Checking techniques have been introduced over the years some are better at proving that the specification does not hold (falsifiers) while others are better at proving that the specification holds (verifiers). These techniques exploit smart data structures, such as Binary Decision Diagrams and Bloom filters, or complicated algorithms such as SAT solvers.

Fig. 3 displays a pseudocode of a Model Checking algorithm. This algorithm performs a breadth first search starting from the initial states of the model ($M.\Sigma_0$) until a bug is found (lines 6–7) or until all the state space was explored (lines 5 and 11). The algorithm introduces two auxiliary sets *Seen* and *Frontier*, which are initialized to the set of initial states of the model. *Seen* contains all the states that were visited in the computation so far and *Frontier* stores the states that were discovered in the last forward step. The algorithm iteratively performs a forward step from the states of *Frontier* (line 8) and removes the states that were already seen so far (line 9). If a

state that violates the specification is found then a message is displayed (lines 6–7) otherwise, the algorithm continues until a fixpoint is reached (lines 5 and 11) and a message that the property holds is displayed.

2.5. Running example

The applicability of our technique is illustrated by Fig. 4, which gives a running example of a program that operates on shared data. Ever 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 shared 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.

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 may be so large that 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.

Fig. 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 while t_1 holds `KeyLock`, C(`numDelItr`) is refined to contain only 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 $\emptyset$. The empty set indicates that there is no consistent lock protecting `numDelItr`, which is a violation of the locking discipline.

```
// 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
```

Fig. 4. A running example.

3. Static datarace detection using Lockset information

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 need to be explored. Our algorithm chooses the particular prefix by exploiting locking information along the trace. Many state-of-the-art dynamic datarace detection tools are lock-based. Therefore, our algorithm is compatible with these tools and can be extended 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 Fig. 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 uses a model checker to find a witness suffix.

Definition 3.1. 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) at least one of a_1 or a_2 is a write operation;
(2) a_2 is the first action after a_1 on Π; and
(3) at least one of a_1 or a_2 is not a protected access event.

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.

	Thread I		Thread II		C(numDelItr)
	operations	*hold_lock*	*operations*	*hold_lock*	
π_1^1	⋮	ψ			Ω
π_1^2			⋮	χ	Ω
π_1^3	synchronized(KeyLock)	{KeyLock}			Ω
π_1^4	if(numDelItr%5==0)	{KeyLock}			{KeyLock}
π_1^5			synchronized(KeyLock)	{KeyLock}	{KeyLock}
π_1^6			if(numDelItr%5==0)	{KeyLock}	{KeyLock}
π_2^1	⋮	ξ			{KeyLock}
—			⋮	η	{KeyLock}
π_2^2	synchronized(DelLock)	{DelLock}			{KeyLock}
π_2^3	++numDelItr	{DelLock}			$\emptyset$

Fig. 5. An execution of Lockset on the running example shown in Fig. 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.

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.

3.1. 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. This is accomplished using 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) that 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 Fig. 6. But it may also indicate a limitation of our current approach, as in Fig. 7. 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, where σ_{a_1} is a global program state that appears right before a_1 on the witness prefix. For this reason, we scan for the last access event that satisfies the above conditions.

Fig. 8 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 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.

In the running example from Fig. 5, the algorithm finds the following prefix: $\pi_1^1 \,.\, \pi_1^2 \,.\, \pi_1^3 \,.\, \pi_1^4 \,.\, \pi_1^5$. This prefix is chosen because operation π_1^6 is the last access event before π_2^3 that fulfills the above conditions with π_2^3.

3.2. 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.

3.2.1. Constructing a model

We construct a model for the program fragment between a_1 and a_2. An alternative method is to first construct a model for the whole program and then reduce its size by eliminating parts which do not occur between a_1 and a_2.

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	

Fig. 6. An example without a datarace. In this example, the algorithm for finding a_1 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 3.1. 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			

Fig. 7. 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.

We can use program chopping techniques [24] to construct a model of the program fragment between a_1 and a_2.

Explanations on building a model from a program are provided in [3]. The model is constructed using an under-approximation—every integer is represented by a constant number of bits. In addition to the program variables, the model includes a program counter pc_i, for each thread of the multi-threaded program, which holds the value of the next operation, and a scheduler *sc* that holds the ID of the next thread that will execute the next operation. One way to build the model is by using guarded commands. A guarded command is a combination of a condition (guard), and a relation whose execution is controlled by the condition, so that the condition guards the execution of the relation. Each guarded command is of the

```
1  ExtendConstructWitnessPrefix(Π, π, V)
2      Let Π be the runtime execution prefix until a1 (exclusive)
3      Let π be the trace from a1 until a2
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^a1
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^a1 ∧ (τ^α = Write ∨ τ^a1 = Write) ∧ (¬ψ^α ∨ ¬ψ^a1))
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
```

Fig. 8. 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 cannot advance until the end of the analysis.

form: $(sc = i \wedge pc_i = PC_1) \Longrightarrow (X \leftarrow f(X, Y, Z) \wedge pc_i \leftarrow PC_2 \wedge sc \leftarrow \{0..i\})$. The guard is always a condition on the values of sc and pc_{sc} and the relation changes the value of sc and pc_{sc}, as well as the value of a variable.

Our algorithm for constructing the model consists of three phases:

- Build a model M of the program. To this model we add a sink state σ_{overflow}, along with transitions $(\sigma, ac, \sigma_{\text{overflow}})$ for every state σ and action ac that cause an overflow at σ. In addition, we add a transition $(\sigma_{\text{overflow}}, \texttt{error}, \sigma_{\text{overflow}})$ for a new action `error` that does not appear in the program.
- For every thread t that takes part in the new model, set the initial values of all the local variables of t to their value at σ_{a_1}, and, in addition, set the initial value of the program counter of t to its last value at σ_{a_1}.
- Set the initial value of each global variable to its value at σ_{a_1}.

3.2.2. Further reduction of the model size

After the model is 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 that 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 3.1 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.

It is important to note that the cost of model checking is reduced by:

- The reduction in model size—focusing on the program fragment between a_1 and a_2. This enables us to employ strong reductions, such as program chopping [24].
- 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 when looking for a datarace.

3.2.3. 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 with a property requiring that a_1 and a_2 be performed one after the other. The model checker performs 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 simply requires that t^{a_2} perform a_2.

The witness suffix that the model checker generates for the datarace on variable `numDelItr` in Fig. 4 is built from operations $\pi_2^1 \,.\, \pi_2^2 \,.\, \pi_2^3$. The following lemmas show that any trace generated by our technique is a valid witness for a datarace. In the proofs of these lemmas we denote the infinite model of the multithreaded program by MP.

Lemma 3.1. *Every program trace π^M in M such that $\sigma_{\text{overflow}} \notin \pi^M$, 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 σ_{overflow} to M, tuples $(\sigma, ac, \sigma_{\text{overflow}})$ to R for every state σ, and an action ac that causes a variable overflow. In addition, we add a tuple $(\sigma_{\text{overflow}}, \texttt{error}, \sigma_{\text{overflow}})$ for a new action `error` that does not appear in the program.
(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 3.2.3 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 σ_{overflow}. Therefore, $R^{M'}|_{\Sigma^{M'}\setminus\sigma_{\text{overflow}}} = \{(\sigma, ac, \sigma') \mid (\sigma, ac, \sigma') \in R^{M'} \wedge \sigma' \neq \sigma_{\text{overflow}}\} \subseteq R^{MP}$, which means that every tuple $(\sigma_i, ac_i, \sigma_{i+1})$ in $R^{M'}|_{\Sigma^{M'}\setminus\sigma_{\text{overflow}}}$ 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 $\sigma_{\text{overflow}} \notin \pi$. From the trace definition we know that each i such that $\sigma_i, ac_i, \sigma_{i+1} \in \pi$ implies that $(\sigma_i, ac_i, \sigma_{i+1}) \in R^{M'}$. σ_{overflow} is not in π, and we know that $R^{M'}|_{\Sigma^{M'}\setminus\sigma_{\text{overflow}}} \subseteq R^{MP}$ 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 3.2. *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 3.3. *If a trace π_2 is returned by the model checker, then $\sigma_{\text{overflow}} \notin \pi_2$.*

Proof. If the model checker returns a trace π_2, then $\pi_2 = \sigma_{a_1}.ac_1 \ldots a_2$. Because a_2 is an action that appears in the program, we know that $a_2 \neq$ `error` and therefore we conclude that a_2 is not enabled at σ_{overflow}. In addition, because $\{\sigma_{\text{overflow}}\} = \{\sigma \mid \exists\, ac\ s.t.\ (\sigma_{\text{overflow}}, ac, \sigma) \in R\}$, we conclude that $\sigma_{\text{overflow}} \notin \pi_2$. □

Corollary 3.1. *Lemmas* 3.1–3.3 *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 ready 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 $\sigma_{\text{post } a_1}$ such that $(\sigma_{a_2}, a_1, \sigma_{\text{post } a_1}) \in R$, and add $\sigma_{\text{post } a_1}$ and a_1 to $\pi_1.\pi_2$ to generate a trace $\pi.\sigma_{a_2}.a_1.\sigma_{\text{post } a_1}.a_2$.

Lemma 3.4 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 3.4. *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 3.1) 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) at least one of a_1 or a_2 is a write operation;
(2) a_2 is the first action after a_1 on Π;
(3) at least one of a_1 or a_2 is not a protected access event.

The algorithm for finding an access event a_1 (Section 3.1) guarantees that:

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

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 3.1 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 $\omega = \pi.\sigma_{a_2}.a_1.\sigma_{\text{post } a_1}.a_2$. 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 $\sigma_{\text{post } a_1}$ such that $(\sigma_{a_2}, a_1, \sigma_{\text{post } a_1}) \in R$. $\pi.\sigma_{a_2}.a_1.\sigma_{\text{post } a_1}$ 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 $\pi.\sigma_{a_2}.a_1.\sigma_{\text{post } a_1}.a_2$ is a program trace prefix in which a_2 is the first action after a_1. Therefore, ω is a witness for a datarace. □

Fig. 9 illustrates the witness that our technique generates for a datarace on variable `numDelItr` in the running example in Fig. 5.

This algorithm can be applied several times on the same trace in order to generate witnesses for different warnings displayed by Lockset. For each warning, Phase 1 chooses a different a_1 and consequently a different prefix is used for generating a wit-

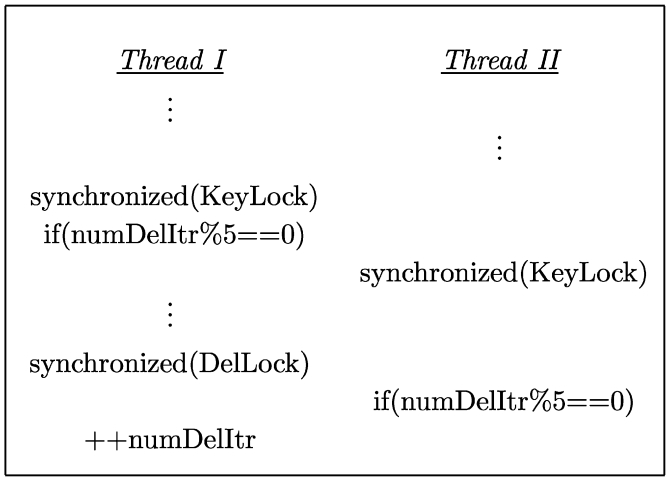

Fig. 9. A witness that our technique generates for a datarace on variable `numDelItr` in the running example in Fig. 5.

ness for a different datarace. However, when several witnesses generated using single dynamic executions then the entire witnesses share some prefix.

Employing a prefix from the dynamic execution significantly ease the work of the model checker in finding dataraces in large programs. However, in some cases, the datarace is not reachable from σ_{a_1} and therefore, using σ_{a_1} as the initial state of the model in these cases cause our model checker to miss dataraces. One possible way of trying to overcome this problem is by running the dynamic tool several times with a nondeterministic scheduler and trying to generate a witness, for each datarace, using few prefixes. This could be done using tools that controls the scheduler behavior such as [13].

4. 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 applied our prototype to several public-domain programs, as well as to other programs that were enhanced to demonstrate the utility of our technique.

4.1. 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. The IBM Watson tool from [8] was used.

The remaining stages are executed per violation warning.

Stage II Finds an additional access event a_1 as described in Section 3.1.

Stage III Reduces the size and depth of the model by generating a longer witness prefix (Section 3.1).

Stage IV Generates a transition system (model) for the model checker (see Section 3.2).

Stage V Applies the model checker in order to generate a trace for the transition system. We used the IBM Haifa tool Wolf [2], which is built on top of RuleBase Parallel Edition [19]. Wolf is a symbolic model checker tuned for software, which uses disjunctive partitioning [3]. 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 3.2.

4.2. Benchmark programs

We used the benchmark programs shown in Table 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 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 nondeterminism. *DQueries* is a similar, but more complicated program.

4.3. 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 Fig. 8, 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 even after a week.

Moreover, in all our modified programs dataraces did not occur during the dynamic execution and therefore, dynamic datarace detection tools based on Lamport happens-before, such as Djit, would miss these dataraces. As expected, Lockset finds violations of the locking discipline even on traces that do not actually have dataraces, and the model checker produces witnesses on all our examples using Lockset information. This shows that our hybrid technique is able to provide witness for dataraces in cases where other techniques fail.

Our tool generated one datarace witness for each benchmark program, and did not produce witnesses for spurious warnings provided by Lockset.

Characteristic of most of the witnesses is a long prefix containing many operations performed by different threads, followed by a shorter suffix containing many operations of t^{a_2}. When we use the reduction suggested in Section 3.2.2, the suffix produced by Wolf lacked operations performed by t^{a_1}. In some cases this reduction produced a larger suffix (hence a larger witness), but still, the runtime with this reduction was significantly better than using t^{a_1} as part of the model. In `our_tsp` we were able to find a witness only when we applied this reduction.

Table 1
Our benchmark programs

Program	Description	Lines	Bits per int
`tsp`	A traveling salesman program from ETH [32]	706	5
`our_tsp`	An enhanced version of `tsp`	708	5
`mtrt`	A multithreaded raytracer from specjvm98 [31]	3751	–
`hedc`	A Web crawler kernel from ETH [32]	29948	6
`SortArray`	A program which performs a parallel sort on an array from [27] A datarace was added	362	7
`PrimeFinder`	A program which finds all the prime numbers in a given interval from [27] A datarace was added	129	8
`Elevsim`	An elevator simulator	150	5
`DQueries`	A shared database simulator	166	4

Table 2
The runtime and memory consumption of the Wolf Model Checker 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	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

Our experimental results were obtained on an Intel Xeon dual CPU 2.4 GHz, 2.5 GB RAM platform running Linux.

The last column of Table 1 shows the number of bits that we use to represent an integer in the transition systems of our benchmark programs.

Table 2 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 3. 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 [6] to direct the model checker's search toward a_2. The hint is very simple and is derived from the Lockset execution. For each model checking 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 that only includes operations of t^{a_2}, this hint has a very strong effect on both runtime and memory consumption.

Table 3 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 three threads, clearly demonstrates the effectiveness of this simple hint. The model checker's runtime decreases from 44695.7 to 2645.57 s and memory consumption decreases from 697 to 143 MB. Another good example is `DQueries` using four threads, which finished after 585.97 s using a hint and timed-out after a week without using a hint.

Table 4 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 drastically increases 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 dataraces without this addition. This demonstrates the effectiveness of our heuristic.

Fig. 10 shows the benefit of removing t^{a_1} from the model and using a hint. Graphs (i)–(iii) compare the techniques presented in Tables 2–4 using two, three, and four threads, respectively. Each bar represents an example. The gray bars represent the runtime of the technique used in Table 3 divided by the runtime of the technique that gave the worst result for the specific example. The white bars represent the same for the technique used in Table 2, and the black bars for the technique used in Table 4. Graph (iv) displays the runtime ratio between a run where a technique that removes t^{a_1} from the model and uses a hint and a run where only a hint was used. The gray, black, and white bars represent two, three, and four threads respectively.

Table 3
The runtime and memory consumption of the Wolf Model Checker 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

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 4
The runtime and memory consumption of the Wolf Model Checker 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

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

The graphs clearly show that removing t^{a_1} from the model and using a hint decreases runtime drastically in all the examples, even when the number of threads is larger than two.

5. Related work

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

5.1. 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 [14]. This tool was expanded in [15] in order to handle large programs. However, it may produce many spurious alarms. Yahav [33] 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. Warlock [28] 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. There are also generic model checking tools which can identify dataraces (e.g., [17]).

5.2. 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 [20], 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 that can occur in a thread interleaving other than the one that was monitored. Savage et al. developed Eraser [26], 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 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 improves the performance of Eraser, but monitoring at the object level causes too many spurious warnings to be displayed. Choi et al. [8] have developed a lock-based dynamic tool. They reduce the number of spurious alarms by using points-to-static analysis and by filtering out all the races that 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

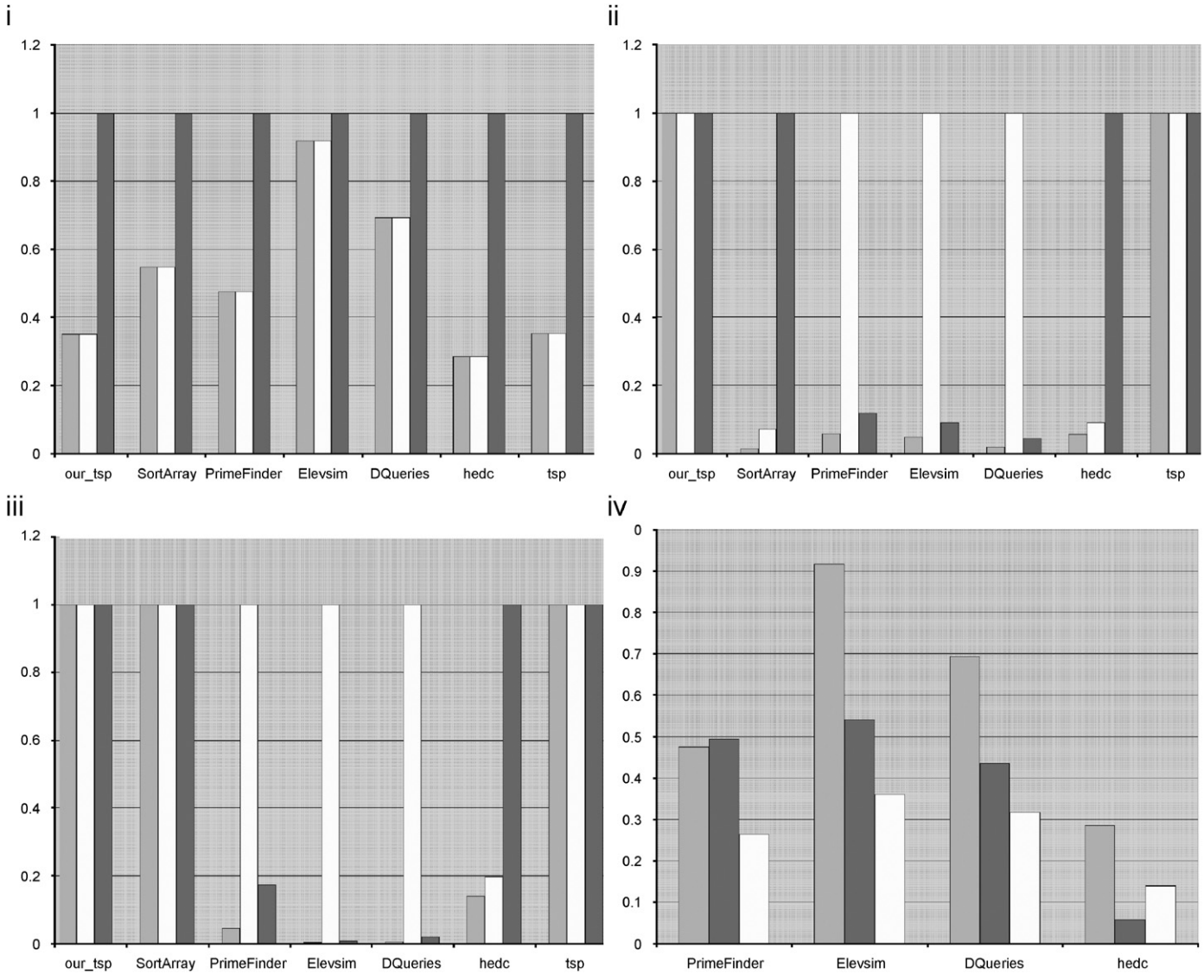

Fig. 10. Results graphs.

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 [10] have developed a hybrid tool from their Lockset-based tool [8] 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.

5.3. Combined static and dynamic tools

Havelund [16] has developed a hybrid tool that combines Lockset and model checking. Havelund's work tries to exploit Lockset warnings to filter out threads from the model. The filtering is done by creating a model that 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.

5.4. 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 that is 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 nondeterministic scheduler is used. Such limitations can be overcome by nondeterministic execution (e.g., [13]). 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.

6. 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 bounded (e.g., CBMC [12]) model checking can be employed.

Acknowledgments

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

References

[1] A. Aiken, D. Gay, Barrier inference, in: Symposium on Principles of Programing Languages, 1998, pp. 342–354.
[2] S. Barner, Z. Glazberg, I. Rabinovitz, Wolf—bug hunter for concurrent software using formal methods, in: Computer Aided Verification, July 2005, pp. 153–157.
[3] S. Barner, I. Rabinovitz, Efficient symbolic model checking of software using partial disjunctive partitioning, in: Advanced Research Working Conference on Correct Hardware Design and Verification Methods, October 2003, pp. 30–35.
[4] A.J. Bernstein, Analysis of programs for parallel processing, IEEE Trans. Electron. Comput. EC-15 (5) (1966) 757–763.
[5] A. Biere, A. Cimatti, E. Clark, Y. Zhu, Symbolic model checking without bdds, in: Tools and Algorithms for the Construction and Analysis of Systems, 1999, pp. 193–207.
[6] R. Bloem, K. Ravi, F. Somenzi, Symbolic guided search for ctl model checking, in: Design Automation Conference, ACM Press, New York, 2000, pp. 29–34.
[7] G. Brat, K. Havelund, S. Park, W. Visser, Model checking programs, in: IEEE International Conference on Automated Software Engineering, 2000.
[8] J.D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, M. Sridharan, Efficient and precise datarace detection for multithreaded object-oriented programs, in: Conference on Programing Language Design and Implementation, 2002.
[9] J.D. Choi, A. Loginov, V. Sarkar, Static datarace analysis for multithreaded object-oriented programs, in: Conference on Programing Language Design and Implementation, 2003.
[10] J.D. Choi, R. O'Callahan, Hybrid dynamic data race detection, in: Symposium on Principles and Practice of Parallel Programming, ACM Press, New York, 2003, pp. 167–178.
[11] E.M. Clarke, O. Grumberg, D.A. Peled, Model Checking, MIT Press, Cambridge, MA, December 1999.
[12] CMU. CBMC ⟨http://www-2.cs.cmu.edu/~modelcheck/cbmc/⟩.
[13] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, S. Ur, Framework for testing multi-threaded java programs, Concurrency and Computation: Practice and Experience 15 (3–5) (2003) 485–499.
[14] C. Flanagan, S. Freund, Type-based race detection for java, in: Conference on Programing Language Design and Implementation, 2000.
[15] C. Flanagan, S.N. Freund, Detecting race conditions in large programs, in: Workshop on Program Analysis for Software Tools and Engineering, ACM Press, New York, 2001, pp. 90–96.
[16] K. Havelund, Using runtime analysis to guide model checking of java programs, in: SPIN, 2000, pp. 245–264.
[17] T. Henzinger, R. Jhala, R. Majumdar, S. Qadeer, Thread-modular abstraction refinement, in: Computer Aided Verification, July 2003, pp. 262–274.
[18] G. Holzmann, The SPIN Model Checker, Addison-Wesley, Reading, MA, 2003.
[19] IBM. Rulebase parallel edition. ⟨http://www.haifa.il.ibm.com/projects/verification/⟩ RB_Homepage/index.html, 2004.
[20] A. Itzkovitz, A. Schuster, O. Zeev-Ben-Mordehai, Towards integration of data race detection in dsm systems, J. Parallel Distrib. Comput. 59 (2) (1999) 180–203.
[21] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Commun. ACM 21 (7) (1978) 558–565.
[22] L. Lamport, How to make a multiprocessor computer that correctly executes multiprocess programs, IEEE Trans. Comput. 28 (9) (1979) 690–691.
[23] E. Pozniansky, A. Schuster, Efficient on-the-fly data race detection in multithreaded C++ programs, in: Symposium on Principles and Practice of Parallel Programming, ACM Press, New York, 2003, pp. 179–190.
[24] T. Reps, G. Rosay, Precise interprocedural chopping, in: Symposium on Foundations of Software Engineering, ACM Press, New York, 1995, pp. 41–52.
[25] A. Roychoudhury, T. Mitra, Specifying multithreaded java semantics for program verification, in: International Conference on Software Engineering, 2002, pp. 489–499.
[26] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, T. Anderson, Eraser: a dynamic data race detector for multithreaded programs, ACM Trans. Comput. Systems 15 (4) (1997) 391–411.
[27] K. Shagin, Benchmark programs, ⟨http://www.cs.technion.ac.il/~konst/benchmarks/⟩.
[28] N. Sterling, Warlock: a static data race analysis tool, in: USENIX Winter Technical Conference, 1993, pp. 97–106.
[29] S.D. Stoller, Model checking multi-threaded distributed java programs, in: SPIN, 2000, pp. 224–244.
[30] R.N. Taylor, Complexity of analyzing the synchronization structure of concurrent programs, Acta Informatica 19 (1983) 57–84.
[31] The Standard Performance Evaluation Corporation, Spec jvm98 benchmarks, 1996 ⟨http://www.spec.org/osg/jvm98⟩.
[32] C. von Praun, T.R. Gross, Object race detection, in: Conference on Object Oriented Programming, Systems, Languages, and Applications, ACM Press, New York, 2001, pp. 70–82.

[33] E. Yahav, Verifying safety properties of concurrent Java programs using 3-valued logic, in: Symposium on Principles of Programing Languages, 2001.
[34] J. Yuan, J. Shen, J.A. Abraham, A. Aziz, On combining formal and informal verification, in: Computer Aided Verification, 1997, pp. 376–387.

Ohad Shacham received his M.Sc. in computer science from Tel Aviv University in 2005 under the supervision of Professor Mooly Sagiv and Professor Assaf Schuster. He received his B.A. degree in Mathematics from Haifa University in 2001.
Ohad Joined IBM at 2000 and he leads the SAT-based formal verification activities at the IBM Haifa research lab.

Mooly Sagiv received his Ph.D. in Computer Science from the Technion Israel Institute of Technology, Haifa. He joined Tel Aviv University's School of computer science as a faculty member in 1997. He has been a visiting professor at the University of Chicago and Datalogisk Institute at the University of Copenhagen. In addition, he has been a researcher at the University of Wisconsin-Madison and IBM's Israel Scientific Center. His research interests include Programming Languages, Compilers, Abstract interpretation, Profiling, Pointer Analysis, Shape Analysis, Interprocedural dataflow analysis, Program Slicing, and Language-based programming environments.

Assaf Schuster (http://www.cs.technion.ac.il/~assaf) received his B.Sc., M.Sc., and Ph.D. degrees in Mathematics and Computer Science from the Hebrew University of Jerusalem. Since being awarded his Ph.D. degree in 1991 he has been with the Computer Science Department at the Technion, The Israel Institute of Technology, where he teaches courses in Operating Systems, Computer Architecture and Parallel and Distributed Computing.