Runtime Analysis of Atomicity for Multithreaded Programs

Liqiang Wang and Scott D. Stoller

Abstract—Atomicity is a correctness condition for concurrent systems. Informally, atomicity is the property that every concurrent execution of a set of transactions is equivalent to some serial execution of the same transactions. In multithreaded programs, executions of procedures (or methods) can be regarded as transactions. Correctness in the presence of concurrency typically requires atomicity of these transactions. Tools that automatically detect atomicity violations can uncover subtle errors that are hard to find with traditional debugging and testing techniques. This paper describes two algorithms for runtime detection of atomicity violations and compares their cost and effectiveness. The reduction-based algorithm checks atomicity based on commutativity properties of events in a trace; the block-based algorithm efficiently represents the relevant information about a trace as a set of blocks (i.e., pairs of events plus associated synchronizations) and checks atomicity by comparing each block with other blocks. To improve the efficiency and accuracy of both algorithms, we incorporate a multilockset algorithm for checking data races, dynamic escape analysis, and happen-before analysis. Experiments show that both algorithms are effective in finding atomicity violations. The block-based algorithm is more accurate but more expensive than the reduction-based algorithm.

Index Terms—Concurrent programming, testing and debugging, Java, data race, atomicity.

1 Introduction

Multithreading has become a common programming technique. Not only operating systems but also many applications are multithreaded. However, developing multithreaded programs is difficult. Concurrency introduces the possibility of errors that do not exist in sequential programs. Furthermore, multithreaded programs may behave differently from one run to another because threads are scheduled indeterminately. For most systems, the number of possible schedules is enormous and testing the system's behavior for each possible schedule is infeasible. Specialized techniques are needed to ensure that multithreaded programs do not have concurrency-related errors.

Threads often communicate by sharing data. Concurrent accesses to shared data should be properly synchronized. Two common errors are deadlocks and data races. A *deadlock* occurs when all threads are blocked, each waiting for some action by one of the other threads. Two accesses to shared variables *conflict* if they access the same variable and at least one of them is a write. Following [28], a *data race* occurs when two concurrent threads perform conflicting accesses and the threads use no explicit mechanism to prevent the accesses from being simultaneous.

Numerous static and runtime (dynamic) analysis techniques are designed to ensure that concurrent programs are free of deadlocks and data races [9], [5], [4], [28], [6]. But, this does not ensure the absence of all synchronization errors. Consider the implementation of `Vector` in Sun JDK 1.4.2, part of which appears in Fig. 1. Consider the following execution of the program at the bottom of Fig. 1: `thread_1` constructs a new vector `v2` from another vector `v1` with k elements by calling the constructor for `Vector`. But, before the constructor completes, `thread_1` yields execution to `thread_2` immediately after statement 1 in the `Vector` constructor. `thread_2` removes all elements of `v1` and then `thread_1` resumes execution at statement 2. The incorrect outcome is that `v2` has k elements, all of which are `null` because the `elementData` array of `v2` is allocated according to the previous size of `v1`. A more subtle error occurs if `thread_2` executes `v1.add(o)` instead of `v1.removeAllElements()`. Then, if $k < 10$, the length of `elementData` is smaller than the new size of `v1`. Although a larger array is allocated in `toArray` to store the elements of `v1`, the array is not returned to the constructor of `v2`, thus `v2` will incorrectly be full of `null` elements. No exception is thrown in these scenarios. Methods `size()`, `toArray(Object[])`, `removeAllElements()`, and `add(Object)` are synchronized, hence there is no data race in these examples.

The incorrect behavior reflects a higher-level synchronization error, namely, lack of atomicity. Atomicity is well-known in the context of transaction processing, where it is sometimes called *serializability*. The methods of concurrent programs are often intended to be atomic. A set of methods is *atomic* if concurrent invocations of the methods are always equivalent to performing the invocations serially (i.e., without interleaving) in some order. The first scenario of the example in Fig. 1 contains two invocations, one of `Vector(Collection)` and one of `removeAllElements()`, which obviously do not have an equivalent serial execution. Therefore, these methods violate atomicity. Similarly, the second scenario also shows a violation of atomicity.

Flanagan and Qadeer developed a type system for atomicity [14]. It can ensure that methods are atomic in all possible executions. However, type inference for the type system is NP-complete [11], [12], so the type system may

• *The authors are with the Computer Science Department, State University of New York at Stony Brook, Stony Brook, NY 11794-4400. E-mail: {liqiang, stoller}@cs.sunysb.edu.*

Manuscript received 11 July 2004; revised 13 Oct. 2005; accepted 12 Dec. 2005; published online 15 Feb. 2006.
Recommended for acceptance by B. Ryder.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0137-0704.

0098-5589/06/$20.00 © 2006 IEEE Published by the IEEE Computer Society

```
public class Vector extends ... implements ... {
 public Vector(Collection c) {
  // c is v1, elementCount is the field of v2.
1 elementCount = c.size();
2 elementData = new Object[(int)Math.min(
       (elementCount*110L)/100,Integer.MAX_VALUE)];
3  c.toArray(elementData);
 }
 public synchronized int size(){return elementCount;}
 public synchronized Object[] toArray(Object a[]) {
  if (a.length < elementCount)
   /* i.e. v2.length < v1.elementCount, this branch
      will be taken if v1.add is executed.*/
   a=(Object[])java.lang.reflect.Array.newInstance(
      a.getClass().getComponentType(),elementCount);
   System.arraycopy(elementData,0,a,0,elementCount);
   if (a.length > elementCount)
     a[elementCount] = null;
   return a;
 }
 public synchronized void removeAllElements() {...}
 public synchronized boolean add(Object o) {...}
}

thread_1
Vector v2 = new Vector(v1);

thread_2
v1.removeAllElements(); or v1.add(o);
```

Fig. 1. An example showing that the constructor of `java.util.Vector` in Sun JDK 1.4.2 violates atomicity.

require manual annotation of the program. Some aspects of a program's behaviors, such as happen-before relations due to start-join on threads, are harder to analyze statically than dynamically and are not considered by the type system because of false alarms (i.e., type errors for methods that are atomic).

This paper presents two runtime algorithms for detecting potential violations of atomicity: a reduction-based algorithm and a block-based algorithm. Runtime analysis is less powerful than static analysis because it cannot ensure correctness of all unexplored behaviors of the system, but may be more precise (i.e., give fewer false alarms) for the explored behaviors. Furthermore, runtime analysis does not require annotations of the code that are often required by type systems; this is a significant practical advantage. Our algorithms do not merely look for violations of atomicity in the observed execution, but also attempt to determine whether the nondeterminism of thread scheduling could allow violations in other executions. We implemented both algorithms. Experiments show that they can successfully find subtle errors.

Our algorithms rely on defaults or information from the user to determine which execution fragments should be considered as transactions. Such user input would typically be provided by annotating some code blocks as expected to be atomic and considering executions of those code blocks as transactions. In either case, our algorithms can automatically check atomicity of the indicated transactions. In contrast, atomicity type systems may require additional help (in the form of type annotations) from the user to determine whether specified code blocks are atomic. Of course, the defaults, whatever they are, will sometimes not capture the user's intentions accurately, so input from the user is desirable, but not always available in practice.

The reduction-based algorithm is an extension of our original reduction-based algorithm [31] and Flanagan and Freund's Atomizer algorithm [10]. It first determines whether there is data race on each variable and then uses this information to classify events. If the sequence of events in each transaction matches a given pattern, then the transactions are atomic.

The block-based algorithm extends our original block-based algorithm [31]. It first constructs blocks (i.e., pairs of events plus associated synchronizations) from an observed trace and then compares each block with all blocks in other transactions. If two blocks are found that violate some given conditions, the transactions containing them are not atomic. The block-based algorithm can infer the atomicity for not only the observed trace, but also all permutations of the trace that are consistent with the synchronization events in the observed trace.

The block-based algorithm is somewhat more expensive than the reduction-based algorithm, but is more accurate, i.e., reports fewer false alarms. This is demonstrated by the experiments in Section 8.

We design a dynamic escape analysis and a happen-before analysis to support the reduction-based algorithm and the block-based algorithm. The dynamic escape analysis dynamically determines when an object escapes from its creating thread, i.e., when it becomes accessible to other threads. The happen-before analysis determines whether two events of different threads are concurrent.

This paper focuses on analyzing Java programs, but the techniques can be extended to other languages. Our system instruments the source code by inserting code that sends events to the monitor during the execution of the instrumented program. The monitor implements both detection algorithms. The reduction-based algorithm can applied *online* (i.e., the analysis is applied during execution of the program and warnings are issued based on the information observed so far) or *offline* (i.e., the analysis is applied after the program terminates and warnings are issued based on the entire execution). The block-based algorithm is offline.

Generally, runtime analysis is unsound compared to static analysis because it depends on the input to the program and may miss errors, especially (but not limited to) errors in unexecuted code. Still, our algorithms are sound in a limited sense. Given a particular execution as input, our dynamic escape analysis and happen-before analysis are conservative and our offline reduction-based algorithm and block-based algorithm are conservative tests for atomicity of execution fragments (called transactions), as defined in Section 2. The online version of the reduction-based algorithm is unsound, as for reason discussed in Sections 5.4 and 5.5.

One direction for future work is to decrease the overhead by using static analysis, as in [6], to show absence of data races or atomicity violations in parts of the program, and to apply runtime analysis only to the other parts. Another direction for future work is to accurately detect atomicity violations in programs that use synchronization mechanisms other than locks.

This paper is organized as follows: Section 2 provides background. Sections 3 and 4 present dynamic escape analysis and happen-before analysis, respectively. Sections 5 and 6 describe the reduction-based algorithm and block-based algorithm, respectively. Section 7 describes instrumentation of the source code. Section 8

contains experimental results. Related work is discussed in Section 9.

2 Background

Informally, an *event* is one step in an execution of a program. This paper considers events that perform the following kinds of operations: read and write escaped variables (i.e., variables that are accessible to multiple threads [7]), acquire and release locks,[1] start and join on threads (corresponding to the methods `start` and `join` of `java.lang.Thread`, respectively), enter and exit from invocations of methods, and the barrier synchronization operations discussed in Section 4. For example, `synchronized`(l) {*body*} in Java indicates two events (in addition to the events performed by the body): acquiring lock l at the entry point and releasing it at the exit point. Two distinct accesses (even using the same operation) to a variable are different events. Let $var(e)$ denote the variable on which e operates. Here, a variable means a storage location, e.g., a field of an object. Two read or write operations *conflict* if they act on the same variable and at least one operation is write.

A *transaction* t is a sequence of events executed by a single thread, denoted $thread(t)$. For example, the sequence of events executed during a method invocation is often considered as a transaction. For nested transactions, our algorithms check only the outmost transactions for atomicity since they subsume the inner transactions.

A *trace* tr is a sequence of events from a set of transactions. The transactions may come from different threads. Given a set T of transactions, a trace for T is an interleaving of events from transactions in T that is consistent with the original order of events from each thread and with the synchronization events. A trace for T must contain all events from transactions in T unless the trace ends in deadlock. A trace is consistent with the synchronization events if the following conditions hold: 1) No lock is held by multiple threads at the same time and 2) the happen-before relation between events based on start/join and barrier synchronizations is respected (this relation is defined in Section 4). In this paper, T is obtained by monitoring an execution. All traces for T can be generated from T based on this definition. Let $traces(T)$ denote all traces for T.

In a trace tr, if a read event e_2 reads the value written by event e_1, we call e_1 the *write-predecessor* of e_2 in tr.

Two traces tr_1 and tr_2 are *equivalent* iff 1) they consist of the same set of transactions, 2) each read event has the same write-predecessor in both traces, and 3) each variable has the same final write event in both traces. This corresponds to view equivalence in transaction processing of database systems [3].

A trace is *serial* if, for each transaction, the events in that transaction form a contiguous subsequence of the trace.

A trace is *serializable* if it is equivalent to some serial trace.

A set T of transactions is *atomic* if every trace for T is serializable. In that case, we will informally say that each transaction is atomic, although, formally, it is atomic with respect to T.

1. This paper considers only the structured locking using `synchronized`, the unstructured locking introduced in JDK 5 is ignored.

A trace that ends in deadlock with some thread in the middle of a transaction is not equivalent to any serial trace. A set T of transactions has *potential for deadlock* if some trace for T ends in deadlock. Our atomicity-checking algorithms assume that T has no potential for deadlock. We detect potential for deadlock using an extension of the goodlock algorithm [17]. Our algorithm reports a warning (of potential deadlock) if two threads acquire two locks, ℓ_1 and ℓ_2, in different orders in concurrent thread periods (as defined in Section 4) without first acquiring some other lock that prevents their attempts to acquire ℓ_1 and ℓ_2 from being interleaved. This algorithm is unsound because it can miss the potential for deadlocks involving three or more threads and locks and it can miss deadlocks due to synchronization other than locks (e.g., `wait` and `notify`). The algorithm can be extended to detect potential for deadlock involving any number of threads [1], but this is more expensive and, we believe, not worthwhile in practice.

3 Dynamic Escape Analysis

This section describes how to determine when an object escapes from its creating thread. Before an object escapes, all operations on it can be ignored when checking atomicity.

We say that an object o' *refers to* an object o if $o'.f == o$ for some field f of o' or if o' is an array and $o'[i] == o$ for some index i. An object o may escape from its creating thread when:

- o is stored in a static field or a field of an escaped object.
- o is an instance of `Thread` (including its subclasses) and o.`start` is called. Note that, if o was created by a constructor with a `Runnable` argument r, then o refers to r, so (by the next rule) r escapes when o starts.
- If o' refers to o and o' escapes, then o escapes. This leads to cascading escape.
- o is passed as an argument to a native method that may cause it to escape.

To indicate whether an object has escaped, a Boolean instance field `escaped` is added to every instrumented class; its initial value is `false`. To detect when an object escapes, we instrument all method calls and all stores to static fields, instance fields, and arrays. When an object escapes, it is marked as escaped by setting its `escaped` field to `true` and all objects to which it refers are marked as escaped (and so on, recursively). The reflection mechanism in Java is used to dynamically find all objects to which a given object refers. When an array escapes, all of its elements are marked as escaped. Since fields cannot be added to Java's built-in array classes, we use a hash table that maps from an array reference to a "shadow" object containing an `escaped` field for the array.

For methods whose bodies are not instrumented, specifically, all native methods and methods of uninstrumented library classes, escape information is maintained conservatively by marking all arguments to those methods as escaped. For some frequently used library classes (in particular, collections and maps), we instead use hand-written wrappers that track escape information more accurately without instrumenting the source code of the library classes. For example, our wrapper for the native method `Vector.add` marks the argument as escaped only

if the target object (i.e., the vector) is escaped. Instrumenting library code is sometimes complicated by dependencies between classes, so we instrument library classes only when specifically testing their own atomicity in the experiments of Section 8. More details about instrumentation appear in Section 7.

Compared with this escape analysis, the technique in [7] is more expensive and more precise since our escape analysis does not consider ownership transfer. Our atomicity checking algorithms could easily use a static escape analysis, such as [27], instead of a dynamic one, which is generally more expensive and more precise [7].

4 Happen-Before Analysis

The execution of a thread is separated into different *periods* by occurrences of synchronization events. A thread period *happens before* another thread period if it must end before the other thread period starts.

Our happen-before analysis tracks only happen-before relationships induced by `start` and `join` on threads and by barrier synchronization. A barrier is a rendezvous point for a specified number n of threads. Once all n threads reach the barrier, these threads may continue executing. This happen-before information is used as described in Sections 5.4 and 6.1. Happen-before relationships induced by `wait` and `notify` could also be analyzed; we do not do this because we believe that `wait` and `notify` are rarely used to achieve atomicity.

An identification number (ID) is assigned to each period of each thread. For an event e, let $tpID(e)$ denote the ID of the thread period in which e was executed. A directed acyclic graph, called *happen-before graph*, with an edge for each ID is used to store the temporal ordering relations between thread periods. pid_1 happens before pid_2 if the edge labeled with pid_2 is reachable from the edge labeled with pid_1. If two thread periods pid_1 and pid_2 are not related to each other by happen-before relations, then we say that they are concurrent, denoted $pid_1 \parallel pid_2$. An event in thread period pid_1 can be concurrent with an event from thread period pid_2 only if pid_1 is concurrent with pid_2. Each node of the graph is labeled with `start`, `join`, barrier, `exit` (which denotes the end of a thread or the program), or `enter` (which denotes the starting point of the program).

When a thread t_1 in period pid_1 calls t_2`.start()` to start another thread t_2, we introduce an ID pid'_1 for the new period of t_1 and an ID pid_2 for the first period of t_2 and we add a `start` node, as shown in Fig. 2. Note that pid_1 happens before pid_2.

When thread t_1 in period pid'_1 calls t_2`.join()` to wait for thread t_2 in period pid_2 to terminate, the ID of t_1 is changed from pid'_1 to pid''_1, as shown in Fig. 2. Note that pid_1, pid'_1, and pid_2 happen before pid''_1 and pid'_1 is concurrent with pid_2.

When a thread reaches a barrier, the thread changes its period ID. In the happen-before graph, we add a node for that barrier. For each participating thread, that node has an incoming edge labeled with the old period ID of the thread and an outgoing edge labeled with the new period ID of the thread. For a barrier node, the thread periods on the incoming edges happen before the thread periods on the outgoing edges.

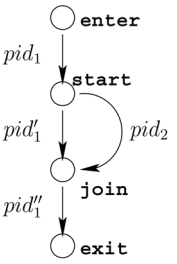


Fig. 2. The happen-before graph for threads t_1 and t_2.

Examining paths in the graph to determine concurrency can be slow when the graph is large, so we cache the results of concurrency queries.

A more efficient but more complicated alternative is to use vector clocks [23], as in [24].

We assume hereafter that start, join, and barrier operations are treated as transaction boundaries, i.e., they separate the preceding events and following events into different transactions and are not contained in any transaction. We adopt this heuristic because execution fragments containing these operations are typically not atomic and, hence, are not expected to be transactions.

5 Reduction-Based Algorithm

In this section, we present an atomicity checking algorithm based on Lipton's reduction theorem [22]. The idea is to infer atomicity from commutativity properties of events.

5.1 Commutativity Properties

Following [22], [15], events are classified according to their commutativity properties. An event is a *right-mover* if, whenever it appears immediately before an event of a different thread, the two events can be swapped (i.e., they can be executed in the opposite order without blocking) without changing the resulting state. A *left-mover* is defined similarly.

For example, if an event e_1 of thread t_1 is a lock acquire, its immediate successive event e_2 from another thread cannot be a successful acquire or release of the same lock because an acquire would block and a release would fail (in Java, it would throw an exception). Hence, e_1 and e_2 can be swapped without affecting the result, so e_1 is a right-mover. Lock release events are left-movers for similar reasons.

An event is a *both-mover* if it is both a left-mover and a right-mover. For example, if there are only read events (no write) on a given variable, the read events commute in both directions with all events, so these read events are both-movers.

Events not known to be left or right movers are *non-movers*.

For Java programs, a classification of events can be conveniently obtained based on synchronization operations. Lock acquire events are right-movers. Lock release events are left-movers. Race-free reads and race-free writes are both-movers [15]. The thread start and join, method enter and exit, and barrier synchronization events are used as transaction boundaries and are not contained in any transaction.

5.2 Reduction-Based Algorithm

Given an arbitrary interleaving of all events in T, if all events of each transaction can be moved together by repeatedly swapping left-movers with the preceding events, right-movers with the subsequent events, and if no trace for T ends in deadlock, then T is atomic because the resulting trace is serial and equivalent to the original trace. If some transaction t contains two or more non-movers, the non-movers could interleave with non-movers in other transactions, preventing the events of transaction t from being moved together. If each transaction t in T has at most one non-mover e, each event in t that precedes e can be moved to the right (toward e) and each event in t that follows e can be moved to the left (toward e), then all events of each transaction can be moved together. These observations motivate the following theorem, where R, L, and N denote right-mover, left-mover, and non-mover, respectively.

Theorem 1. *A set T of transactions is atomic if T has no potential for deadlock and each transaction in T has the form $R^*N^?L^*$.*

Proof. This is a simple variant of Lipton's reduction theorem [22]. □

This theorem, together with a technique for runtime detection of data races (such as the lockset algorithm in [28]), leads directly to an efficient runtime algorithm for checking atomicity. But, this algorithm reports false alarms in several cases. The following sections show how to improve it.

This algorithm for runtime checking atomicity was first proposed in [31] and then [10] and is regarded as a runtime analogue of Flanagan and Qadeer's atomicity type system [14].

5.3 Improvement 1: Read-Only and Thread-Local Variables

A variable is *thread-local* if it is accessed by a single thread. A variable is *read-only* if it is never written, except for the initialization when it is allocated. Accesses to thread-local and read-only variables are race-free and, hence, are both-movers [10], [16].[2] The following theorem treats an entire synchronization block that contains only read-only or thread-local accesses as a both-mover. This is similar but not identical to ideas in [10] and [16], which treat thread-local and protected locks specially, as described below in Section 5.5. The improvement expressed in the following lemma and theorem makes no assumption about the lock being acquired and released, except that acquiring lock does not lead to potential for deadlock.

Let $AcqRel$ denote an acquire of some lock immediately followed by a release of the same lock. Let $AcqA^*Rel$ denote an acquire of some lock, then followed by accesses to read-only or thread-local variables, finally followed by release of the same lock.

Lemma 1. *Given a set T of transactions, T is atomic if T has no potential for deadlock and each transaction in T has the form $(R + AcqRel)^*N^?(L + AcqRel)^*$.*

2. Although the Java Memory Model allows the initializing write to be involved in a data race [26], we do not consider this or other issues raised by Java's controversial weak memory model.

Proof. Based on Theorem 1, it suffices to argue that $AcqRel$ can be ignored when determining atomicity. The only effect that $AcqRel$ could have is to cause a deadlock. This is avoided by the requirement that T has no potential for deadlock. Thus, $AcqRel$ has no effect on the state of the program and the commutativity properties of other operations (e.g., it does not affect whether any accesses to variables are race-free). □

Theorem 2. *A set T of transactions is atomic if T has no potential for deadlock and each transaction in T has the form $(R + AcqA^*Rel)^*N^?(L + AcqA^*Rel)^*$.*

Proof. This follows from Lemma 1 and the fact that events in A commute with all events from other threads, so they have no effect on atomicity. □

Online (i.e., during execution of the program) classification of variables as read-only or thread-local is based on whether the variable has been read-only or thread-local so far; thus, the classification of a variable may change afterward. Offline (i.e., after the program terminates) classification is based on the entire execution and is therefore more accurate.

This improvement can be viewed as synchronization elimination since the idea is to ignore synchronization operations that do not affect the behavior of the program. Prior work on synchronization elimination, such as [27], is generally based on static analysis and intended for optimization, while we use this idea in a runtime analysis to reduce false alarms in program checking. Also, the particular case of synchronization elimination described here has not been considered in the static context to the best of our knowledge.

5.4 Improvement 2: Multilockset Algorithm for Runtime Race Detection

To classify read and write events as both-movers or non-movers, we need to determine whether there is a data race involving these events. This section briefly reviews related work on runtime race detection and then proposes a more precise (and more expensive) algorithm. Naturally, more precise race detection allows more precise reduction-based atomicity checking.

The Eraser algorithm [28], also called the lockset algorithm, is a classic runtime race detection algorithm based on the policy that each shared variable should be protected by a lock that is held whenever the variable is accessed. The algorithm works as follows: For each variable x, a set $lockset(x)$ of locks is maintained. A lock l is in $lockset(x)$ if every thread that has accessed x was holding l at the moment of access. $lockset(x)$ is initialized to contain all locks. Let $locksHeld(t)$ denote the set of locks currently held by thread t. When a thread t accesses x, the lockset is refined (updated) by $lockset(x) := lockset(x) \cap locksHeld(t)$, except during the initialization period, when x is assumed to be accessible only by the thread that allocated it and the lockset retains its initial value. Savage et al. [28] supposes that the initialization period ends when the variable is accessed by a second thread; this is a heuristic that may cause the algorithm to miss some races, but it is easy to implement. When $lockset(x)$ becomes empty, it means that no lock protects x. At that time, if there have been writes to x after the

initialization period for x (hence, x is not read-only), a warning is issued, indicating a potential data race. To see why this treatment of initialization may miss races, consider four consecutive events by two concurrent threads, t_1 and t_2: x is allocated in t_1, then escapes to be accessible to t_2 (note that t_2 does not actually access x yet), and then is accessed by t_1 and then t_2 without holding any locks; a data race occurs, but this algorithm does not report it.

Von Praun and Gross [29] modify the lockset algorithm by introducing a more sophisticated condition for determining when initialization ends. They suppose that, when a variable is accessed by a second thread, its ownership is also transferred. Thus, $lockset(x)$ is not refined until a "third" thread (possibly the same as the first thread) accesses x. This algorithm may miss even more races than the original lockset algorithm. On the positive side, it may produce fewer false alarms. For efficiency, [29] treats an entire object (instead of a field of an object) as a single variable. This reduces the number of maintained locksets, but increases the number of false alarms.

Flanagan and Freund [10] improve the lockset algorithm to avoid false alarms in multiple-reader, single-writer scenarios. For each variable, a pair of locksets is used instead of one lockset: The *access-protecting* lockset contains locks held on every access (read or write) to the variable, and the *write-protecting* lockset contains locks held on every write to the variable. The two locksets for a variable are updated based on their definitions at each access to that variable. A read event on a variable x is race-free if the current thread holds at least one of the write-protecting locks for x; otherwise, a potential data race is reported. A write event on a variable x is race-free if the access-protecting lockset of x is not empty; otherwise, a potential data race is reported.

We propose the *multilockset algorithm*, which is more accurate than the preceding algorithms, i.e., it misses fewer races and reports fewer false alarms. It incurs higher overhead, but is still practical according to the experiments in Section 8. The three main improvements are: 1) The algorithm uses a dynamic escape (from thread) analysis, described in Section 3, to determine when "initialization" of a variable ends, i.e., when to start refining the variable's lockset. This improves accuracy because only the accesses before the variable escapes are ignored. 2) The happen-before relation based on `start` and `join` operations on threads and barrier operations is considered. 3) The analysis maintains multiple read-protecting locksets besides a write-protecting lockset. The access-protecting lockset used in [10] is equivalent to maintaining only one read-protecting lockset and this can cause some false alarms, as illustrated below.

For each variable x, we maintain:

- $ReadSets(x)$, which contains $\subseteq$-minimal sets of held locks for read events on x. In other words, for each read of x, we insert $locksHeld(t)$ in $ReadSets(x)$ and, then, if $ReadSets(x)$ contains S_1 and S_2 such that $S_1 \subseteq S_2$, we remove S_2.
- $WriteSet(x)$, which is the set of locks held on all writes to x, i.e., for the first write, $WriteSet(x) := locksHeld(t)$ and, for each subsequent write to x, $WriteSet(x) := WriteSet(x) \cap locksHeld(t)$.
- $ReadThreadSet(x)$, which contains the IDs of thread periods involving read events on x.

```
if (x is not escaped)
  return no-race-so-far;
/*update protecting locksets and thread period sets
  for each escaped variable access */
if (e is read){
  if (∄ls ∈ ReadSets(x). ls ⊆ locksHeld(t)){
    ReadSets(x) := (ReadSets(x) − {ls|ls ∈ ReadSets(x)∧
                    ls ⊇ locksHeld(t)}) ∪ {locksHeld(t)}
  }
  ReadThreadSet(x) := ReadThreadSet(x) ∪ {t}
} else { /* e is write */
  WriteSet(x) := WriteSet(x) ∩ locksHeld(t)
  WriteThreadSet(x) := WriteThreadSet(x) ∪ {t}
}
/* check whether data race occurs */
switch (WriteSet(x)) {
  case uninitialized: /* no write to x so far. */
    return no-race-so-far;
  case empty:
    /* there is no common lock held on all writes to x*/
    if (IsConcurrent(WriteThreadSet(x)) or
        IsConcurrentWith(WriteThreadSet(x),
                         ReadThreadSet(x)))
      return potential-race;
    else return no-race-so-far;
  case non-empty:
    if ∃ls ∈ ReadSets(x).
      ((ls ∩ WriteSet(x) == ∅) and
        IsConcurrentWith(WriteThreadSet(x),
                         ReadThreadSet(x)))
      return potential-race;
    else return no-race-so-far;
}
```

Fig. 3. The pseudocode of the multilockset algorithm.

- $WriteThreadSet(x)$, which contains the IDs of thread periods involving write events on x.

$ReadSets(x)$, $WriteSet(x)$, $ReadThreadSet(x)$, and $WriteThreadSet(x)$ are not updated by accesses to x before x escapes. The happen-before analysis described in Section 4 determines whether two thread periods can happen concurrently. If there are no concurrent thread periods inside $WriteThreadSet$ or between $ReadThreadSet$ and $WriteThreadSet$, i.e., there are no concurrent conflicting accesses according to the happen-before analysis, the variable must be free of data race. For sets P_1 and P_2 of thread period IDs, $isConcurrentWith(P_1, P_2)$ returns true if some thread period in P_1 is concurrent with some thread period in P_2. When P_1 or P_2 is empty, $isConcurrentWith(P_1, P_2)$ returns false. $isConcurrent(P_1)$ returns true if P_1 contains concurrent thread periods. When P_1 is empty, $isConcurrent(P_1)$ returns false. The pseudocode for the multilockset algorithm is shown in Fig. 3. The first case (i.e., $WriteSet(x)$ is uninitialized) implies that there is no write to x so far, hence no data race. The second case (i.e., $WriteSet(x)$ is empty) uses a conservative test: A potential data race is reported if there is no common lock held on all writes to x and a write and another access to x occur in concurrent thread periods. The third case (i.e., $WriteSet(x)$ is nonempty) also uses a conservative test: A potential data race is reported if some common locks are held at all writes, none of those locks are

```
t1:                          t4:
synchronized(o1){            read(x);
 synchronized(o2){           synchronized(o1){
   write(x);                   synchronized(o2){
   write(x);                     write(x);
 }                             }
}                            }

t2:                          t3:
synchronized(o2){            synchronized(o1){
  read(x);                     read(x);
}                            }
```

Fig. 4. An example to illustrate the accuracy of the multilockset algorithm.

held at some read, and there are at least two concurrent thread periods such that one performs a write to x and another performs a read to x.

According to the experiments in Section 8, this algorithm is practical because its storage requirement is reasonable compared with that of the Eraser algorithm, and their runtime overheads are similar.

This algorithm is more accurate than previous lockset-based algorithms. For example, suppose x has escaped from its creating thread and the threads in Fig. 4 execute in the order t_3, t_2, t_1 (ignore t_4 for now). According to the definition of data race, there is no data race on x. The algorithms in [28], [29], [10], [6] all report a false alarm (potential data race on x). The multilockset algorithm does not.

Even the multilockset algorithm produces some false alarms. For example, consider the threads t_2, t_3, and t_4 in Fig. 4 (ignore t_1 now). If the execution order is t_3, t_2, t_4 since

$$\begin{aligned} &ReadSets(x) = \{\emptyset\}, \\ &WriteSet(x) = \{o_1, o_2\}, \\ &ReadThreadSet = \{t_2, t_3, t_4\}, \end{aligned}$$

and $WriteThreadSet = \{t_4\}$, the third case "nonempty" in Fig. 3 is matched, a potential data race is reported, but this is a false alarm, and it causes a false alarm in atomicity checking. In Section 6, we will see that the block-based algorithm does not produce such a false alarm for this example.

5.5 Other Improvements

We refined the classification of all lock acquires and releases as right-movers and left-movers, respectively, in Section 5.1. In the following cases, they are classified as both-movers [10], [16]. 1) *Reentrant locks*: If the thread already holds the lock, an acquire and the corresponding release on the same lock are both-movers because they have no effect on the execution of the program. 2) *Thread-local locks*: If a lock is used by only one thread, acquire and release on it are both-movers. 3) *Protected locks*: Lock l_2 is protected by lock l_1 if, whenever a thread holds l_2, it also holds l_1. Acquire and release by a thread t on a protected lock l_2 are both-movers because adjacent operations of other threads cannot be operations on l_2 (because t holds l_1).

Offline algorithms can classify thread-local locks and protected locks more accurately than online algorithms. For example, if a lock is protected for a while, but is unprotected later, acquire and release operations on the lock that precede this change will be wrongly classified as both-movers by online algorithms. This may cause atomicity violations to be missed.

The above improvements can be viewed as synchronization elimination and static analyses that recognize these situations have been used for program optimization [27]. We follow the approach in [10] to recognize them using runtime analysis.

5.6 Implementation of Reduction-Based Algorithm

In practice, many of the sets of locks manipulated by the lockset algorithm have size 0 or 1. To save space and time, each lockset is represented by a structure that contains `null` (if the lockset is empty), a direct reference to the element (if the lockset has size 1), or a collection (if the lockset has size greater than 1). Intersection operations could be optimized by maintaining the contents of each set in sorted order, but we did not implement this because most locksets are small.

Our online reduction-based algorithm is implemented following the design in [10] which does not use the transaction tree structure described below. Our implementation of the offline reduction-based algorithm is described next.

We instrument the program by a source-to-source transformation. The instrumented program constructs a tree structure for each transaction during execution. The root corresponds to the entire transaction. Each node other than the root corresponds to a synchronized block (i.e., an execution of a synchronized statement or synchronized method) and is labeled with the acquired lock. The tree structure reflects the nesting of synchronized blocks. In other words, if an execution σ' of a synchronized block is nested inside the execution σ of a synchronized block, the node for σ' is a descendant of the node for σ in the tree structure. Each node is also labeled with the set of variables accessed only once (denoted $varsOne$) and with the set of variables accessed multiple times (denoted $varsMul$) in the corresponding synchronized block, ignoring accesses in subblocks because we need to distinguish the two cases for non-movers, i.e., at most one non-mover or multiple non-movers, if some accesses in the current blocks have data races. Obviously, $varsOne$ and $varsMul$ are disjoint.

For example, Fig. 5 shows the tree structure for the `Vector` example in Fig. 1. The four fields of each node contain the acquired lock (none in the root node), pointers to child nodes, the set of variables accessed only once (i.e., $varsOne$), and the set of variables accessed multiple times (i.e., $varsMul$), respectively. In this example, there is no access outside the three synchronization blocks, hence, the last two fields of both root nodes contain empty sets.

The atomicity of a transaction is determined as follows:

1. Determine the commutativity type of accesses to each variable in $varsOne$ and $varsMul$ at each node of the tree; the accesses are both-mover if accesses to the variable are race-free; otherwise, they are non-mover.
2. For each node, construct the pattern (of commutativities of events represented by that node) by concatenating, in any order, the commutativity types of the variables in the node's $varsOne$ and $varsMul$ sets and, if the node represents a synchronization operation, adding an R at the beginning of the pattern and adding a L at the end of the pattern. For

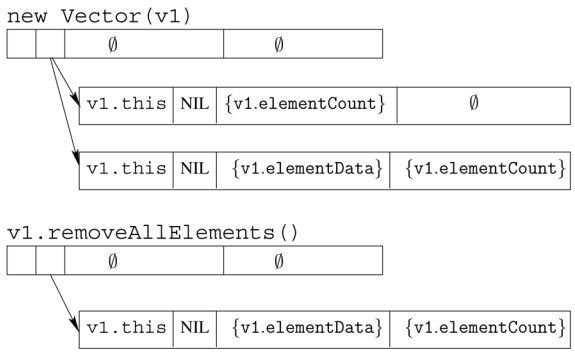


Fig. 5. Tree structure for the `Vector` example of Fig. 1.

each variable in $varsOne$, its commutativity type appears once in the pattern; for each variable in $varsMul$, its commutativity type appears twice.

3. Construct the pattern of commutativities for the transaction by concatenating the pattern for each node during a traversal of the tree (e.g., in-order traversal), except that the R and L representing acquire and release at a node are positioned before and after, respectively, the rest of the pattern constructed from the accesses represented by that node and its descendants.
4. Check whether the pattern of commutativities for the transaction matches the regular expression in Theorem 2. Note that the tree structure does not indicate the relative order of the accesses represented by a node and its descendants and the result of matching against the regular expression in Theorem 2 is insensitive to that order, so any traversal order (in-order, preorder, or postorder) may be used in Step 3.

In Fig. 5, the transaction for `new Vector(v1)` has the pattern $RBLRBBBL$, which does not match the atomicity in Theorem 2; the transaction for `v1.removeAllElements()` has the pattern $RBBBL$. Note that `v1.elementCount` is saved in $varsMul$, hence, it is represented by two both-movers in the patterns constructed for each transaction.

Although the reduction-based algorithm is efficient for checking atomicity, it produces numerous false alarms in the experiments in Section 8. This motivates us to design the novel and more accurate approach discussed next.

6 Block-Based Algorithm

The block-based algorithm determines whether a violation of atomicity is possible in traces obtained from the observed trace by permuting the order of events consistent with the synchronization events. Explicitly computing these permutations would be prohibitively expensive. Our approach is to look for unserializable patterns of operations from these events. We first present an algorithm that works for the case of multiple transactions that share exactly one variable (note that locks and barriers are not counted as shared variables), then extend the ideas in it to handle the case of two transactions that share multiple variables. Finally, we extend that algorithm to handle the case of multiple transactions that share multiple variables.

6.1 Multiple Transactions that Share Exactly One Variable

Given a set T of transactions, the algorithm looks for unserializable patterns of operations of T. An *unserializable pattern* is a sequence in which operations from different transactions are interleaved in an unserializable way. If the transactions of T share exactly one variable, the following unserializable patterns are checked:

- A read in one transaction occurs between two writes in another transaction.
- A write in one transaction occurs between two reads in another transaction.
- A write in one transaction occurs between a write and a subsequent read in another transaction.
- The final write in one transaction occurs between a read and a subsequent write in another transaction.

Note that all of the operations in the patterns are on the same variable (the single shared variable). These patterns can be drawn as follows, where each line corresponds to a transaction and time advances from left to right:

$R(x)$ $W(x)$ $W(x)$	$W(x)$ $R(x)$ $R(x)$
$W(x)$ $W(x)$ $R(x)$	$FW(x)$ $R(x)$ $W(x)$

Informally, T is atomic if no trace for T contains a subsequence that matches any of these patterns; this idea is formalized in Theorem 3 below.

The block-based algorithm looks for these unserializable patterns by considering pairs of "blocks" from different transactions. Intuitively, a block captures the information about two events of the same transaction that is relevant to atomicity checking. Many pairs of events in a transaction may generate the same block. Our algorithm recognizes this and stores only one copy of it. This can eliminate a significant amount of redundant storage and processing during atomicity checking. If the two events operate on the same variable, the block is called a *1v-block*; if the two events operate on the different variables, the block is called a *2v-block*. This section discusses only 1v-blocks; 2v-blocks are discussed in Section 6.2. An access to a variable that has not yet escaped is not used to form blocks.

Specifically, for two events e_1 and e_2 in transaction t with $var(e_1) = var(e_2)$ (call the variable v), a 1v-block is generated for e_1 and e_2 if 1) v is escaped when e_1 and e_2 occur and 2) one of the following conditions holds:

a. If t contains a write to v that precedes e_2, then e_1 is the last write to v that precedes e_2 in t; otherwise, if t contains a read of v that precedes e_2, then e_1 is the last read of v that precedes e_2 in t.
b. If e_2 is the final write to v in t, then e_1 is an initial read of v in t.

An *initial read* of a variable v in a transaction t is a read of v that is not preceded by a write to v in t. Note that the concepts of initial read and final write are relative to the current transaction, not to the entire execution. If there is only one event in a transaction, a dummy event is added. This dummy event is used only for constructing blocks, not for matching part of an unserializable pattern.

If e_1 and e_2 satisfy one of these conditions, then the *1v-block* for e_1 and e_2 is a tuple

$$\langle v, op(e_1), op(e_2), isFW(e_1), isFW(e_2), pid, held(e_1), held(e_2), held(e_1, e_2)\rangle.$$

The notations used for each element are explained next. The first element, v, is the variable on which e_1 and e_2 operate (recall that $var(e_1) = var(e_2)$). The second and third elements, $op(e)$, is the operation type, namely, R (for "read"), W (for "write"), or *dummy*. The fourth and fifth elements, $isFW(e)$, are a Boolean value indicating whether e is the final write on v in t. The sixth element, pid, identifies the thread period in which e_1 and e_2 were executed, i.e., $pid = tpID(e_1) = tpID(e_2)$; recall from Section 4 that each transaction occurs within a single thread period.[3] The seventh and eighth elements, $held(e)$, are the set of locks held by the thread when executing event e. The last element, $held(e_1, e_2)$, is the set of locks held continuously from e_1 to e_2.

For example, the transaction

$$t : acq(\ell_1)\ R(v)\ acq(\ell_2)\ W(v)\ R(v)\ rel(\ell_2)\ rel(\ell_1) \quad (1)$$

has two 1v-blocks, where pid is the thread period ID of t.

$$\begin{aligned} b_1 &: \langle v, R, W, \text{false}, \text{true}, pid, \{\ell_1\}, \{\ell_1, \ell_2\}, \{\ell_1\}\rangle \\ b_2 &: \langle v, W, R, \text{true}, \text{false}, pid, \{\ell_1, \ell_2\}, \{\ell_1, \ell_2\}, \{\ell_1, \ell_2\}\rangle. \end{aligned} \quad (2)$$

To determine whether the operations in two blocks can form an unserializable pattern, we need to determine whether an operation of one block can occur between the two operations of another block. This is determined by locking and happen-before analysis. For 1v-blocks

$$\begin{aligned} b &= \langle v, op_1, op_2, fw_1, fw_2, pid, h_1, h_2, h_{12}\rangle \text{ and} \\ b' &= \langle v, op'_1, op'_2, fw'_1, fw'_2, pid', h'_1, h'_2, h'_{12}\rangle, \end{aligned}$$

an operation op_i ($i \in \{1, 2\}$) of b can occur between operations op'_1 and op'_2 of b', denoted *can-occur-between* ($\langle op_i, h_i, pid\rangle, \langle op'_1, op'_2, h'_{12}, pid'\rangle$), if the condition $(h_i \cap h'_{12} = \emptyset) \wedge (pid \parallel pid')$ is satisfied.

This simple test is accurate provided there is no potential for deadlock in the set of transactions. So, we check potential for deadlock, as described in Section 5.6, as part of the block-based algorithm. To see that this test may be inaccurate if there is potential for deadlock, note that $op(e)$ cannot occur between $op(e_1)$ and $op(e_2)$ in the following example, even though the can-occur-between condition defined above is satisfied. This indicates that ignoring deadlock would lead to more false alarms.

$$\begin{aligned} t &: acq(l_1)\ acq(l_2)\ rel(l_2)\ e\ rel(l_1) \\ t' &: acq(l_2)\ acq(l_1)\ rel(l_1)\ e_1\ e_2\ rel(l_2). \end{aligned} \quad (3)$$

As mentioned above, many pairs of events may produce the same 1v-block. For example, only one 1v-block is generated for the following transaction:

$$W(v)\ R(v)\ R(v)\ R(v). \quad (4)$$

Two 1v-blocks, b and b', are *atomic* with respect to each other, denoted $isAtomic1vBlk(b, b')$, if the synchronization prevents the unserializable patterns described above, i.e., the two operations from one block together with an operation from the other block cannot form one of those unserializable patterns. Formally, $isAtomic1vBlk(b, b')$ holds iff, for all combinations of three operations op_1, op_2, and op_3, where op_1 and op_2 are from one block, op_3 is from the other block, either op_3 cannot occur between op_1 and op_2 or the sequence $op_1\ op_3\ op_2$ does not match any of the unserializable patterns. Obviously, $isAtomic1vBlk(b, b')$ is symmetric. For example, consider a 1v-block,

$$b' = \langle v, W, dummy, \text{true}, \text{false}, pid', \emptyset, \emptyset, \emptyset\rangle$$

in a different transaction t' than transaction t in (1); if $pid \parallel pid'$, then

$$isAtomic1vBlk(b_1, b')$$

and

$$isAtomic1vBlk(b_2, b')$$

do not hold because the unserializable patterns can be formed. For another example, consider a 1v-block

$$b'' = \langle v, R, dummy, \text{false}, \text{false}, pid'', \emptyset, \emptyset, \emptyset\rangle;$$

if $pid'' \parallel pid$, then

$$isAtomic1vBlk(b_1, b'')$$

and

$$isAtomic1vBlk(b_2, b'')$$

hold.

Let $1v\text{-}blocks(t)$ denote the set of 1v-blocks for a transaction t. To check the atomicity of multiple transactions that share exactly one variable, we have the following lemma.

Lemma 2. *Let t and t' be transactions that share exactly one variable with $thread(t) \neq thread(t')$ and suppose they do not have potential for deadlock. $\{t, t'\}$ is atomic iff $\forall b \in 1v\text{-}blocks(t).\ \forall b' \in 1v\text{-}blocks(t').\ isAtomic1vBlk(b, b')$.*

Proof. For the forward implication ($\Rightarrow$), we prove the contrapositive, i.e., if $isAtomic1vBlk(b, b')$ is false for some pair of 1v-blocks b and b', then t and t' are not atomic. This follows easily from the definition of $isAtomic1vBlk$.

For the reverse implication ($\Leftarrow$), suppose $isAtomic1vBlk(b, b')$ holds for all pairs of 1v-blocks b and b'. Let S be a nonserial trace for $\{t, t'\}$. If neither transaction performs a write, then S is obviously equivalent to a serial trace. Suppose, without loss of generality, that t performs the final write e^t_{FW} in S. There are two cases:

Case 1. If t' does not read the value written by e^t_{FW}, then all reads and writes in t' precede e^t_{FW} in S and we can show that S is equivalent to the serial trace S' in which t' precedes t. The main point is that there is no read event $e^{t'}_R$ that reads the value written by any write e^t_W in S because, if there were, then e^t_W and e^t_{FW} would form a block b that can be interleaved in an unserializable way with $e^{t'}_R$, so $isAtomic1vBlk(b, b')$ would be false for some block b' containing $e^{t'}_R$, a contradiction. Similarly, we can show that each read event of t reads the value

3. For efficiency, in our implementation, multiple 1v-blocks that differ only in the thread period IDs are represented by a single 1v-block that contains a set of thread period IDs.

written by the same write event in S' and S. According to the definition for equivalence of traces in Section 2, S is equivalent to S'.

Case 2. If t' reads the value written by e^t_{FW}, then we can show that all reads and writes in t' appear after e^t_{FW} in S (because, if one of those events precedes e^t_{FW}, an unserializable pattern and, hence, a nonatomic pair of 1v-blocks would exist) and that S is equivalent to the serial trace in which t precedes t'. □

Theorem 3. *Let T be a set of transactions that share exactly one variable. Suppose T does not have potential for deadlock. T is atomic iff for all t, t' in T with $thread(t) \neq thread(t')$,*

$$\forall\, b \in 1v\text{-}blocks(t).\forall\, b' \in 1v\text{-}blocks(t').isAtomic1vBlk(b, b').$$

Proof. For the forward implication ($\Rightarrow$), the proof is straightforward, except for details related to final writes.

We prove the reverse implication ($\Leftarrow$) by induction on the number of transactions in T. Let S be a nonserial trace for T. For $T' \subseteq T$, let $S|T'$ denote the subsequence of S containing only events from transactions in T'. Let t be the transaction that performs the final write e^t_{FW} in S. Let T_2 be the set of transactions in T other than t that read the value written by e^t_{FW}. No read or write from T_2 can precede e^t_{FW} in S (otherwise an unserializable pattern would be formed). This implies that T_2 contains no writes (otherwise e^t_{FW} would not be the final write). Thus, $S|T_2$ is equivalent to some serial trace S_2. Let T_1 be $T - T_2 - \{t\}$. For all $t_1 \in T_1$ with $thread(t) \neq thread(t_1)$, the hypothesis of the contrapositive case and Lemma 1 imply that $\{t, t_1\}$ is atomic; since t_1 does not read t's write and t performs the final write in S, t_1 must precede t in every serial trace equivalent to $S|\{t, t_1\}$. Since t can be serialized after every transactions in T_1, S is equivalent to $(S|T_1) \cdot t \cdot S_2$, where the dot denotes concatenation. By the induction hypothesis, $S|T_1$ is equivalent to some serial trace S_1. Thus, S is equivalent to the serial trace $S_1 \cdot t \cdot S_2$. □

Let E be the total number of events in all transactions of T. Let P denote the number of thread periods; P is generally very small except when there are many calls to barrier operations. Rule A for constructing 1v-blocks generates at most $O(E)$ 1v-blocks because it combines each event with at most one preceding event. Rule B constructs at most $O(E)$ 1v-blocks because there is only one final write for each variable in each transaction. Hence, the number of 1v-blocks is $O(E)$. The cost of checking can-occur-between for a pair of 1v-blocks is $O(P)$. Assuming $|locksHeld(t)|$ is always bounded by a constant for all threads t, the worst-case running time of the algorithm, based on Theorem 3, is $O(PE^2)$.

6.2 Two Transactions that Share Multiple Variables

To check the atomicity of two transactions that share multiple variables, the test embodied in Theorem 3 needs to be strengthened.

Consider two events from transaction t and two events from transaction t'. If they operate on four or three different variables, they cannot cause a violation of atomicity. If they all operate on the same variable, the analysis in Section 6.1 applies. Suppose they operate on two variables. If they contain no conflicting operations, or exactly one pair of conflicting operations, they do not cause a violation of atomicity. Suppose they contain two pairs of conflicting operations. We can check based on the definition of atomicity in Section 2 whether every feasible interleaving of the operations from the four events is serializable; if so, the two blocks are atomic. A few illustrative cases of unserializable interleavings are listed in the following table:

0 read	$W(x)$ $W(y)$ $W(x)$ $W(y)$	$W(x)$ $W(y)$ $W(y)$ $W(x)$
1 read	$R(x)$ $W(y)$ $W(y)$ $W(x)$	$R(x)$ $W(y)$ $W(y)$ $W(x)$
2 reads	$R(x)$ $W(y)$ $W(x)$ $R(y)$	$R(x)$ $R(y)$ $W(y)$ $W(x)$

Let $IR(t)$ and $FW(t)$ be the sets of initial reads and final writes, respectively, on shared variables in t. For events e_1 and e_2 of the same thread, let $heldmid(e_1, e_2)$ be the set of locks acquired by that thread after e_1 and released by it before e_2 and not contained in $held(e_1) \cup held(e_2)$; furthermore, reacquires of held locks are ignored when computing $heldmid$.

A *2v-block* for a transaction t is a tuple,

$$\langle var(e_1), var(e_2), op(e_1), op(e_2), pid, held(e_1), held(e_2), held(e_1, e_2), heldmid(e_1, e_2)\rangle,$$

formed from two (read or write) events e_1 and e_2 of t such that e_1 precedes e_2 in t, $var(e_1) \neq var(e_2)$, and e_1 and e_2 are in $IR(t) \cup FW(t)$, where $pid = tpID(e_1) = tpID(e_2)$. Let *2v-blocks(t)* denote the set of *2v-blocks* for transaction t. For example, for the following transaction t,

$$R_1(x)\ W_2(y)\ W_3(x)\ W_4(y)\ R_5(x), \tag{5}$$

$IR(t) = \{R_1(x)\}$, $FW(t) = \{W_3(x), W_4(y)\}$, and *2v-blocks(t)* contains $\langle x, y, R, W, pid, \emptyset, \emptyset, \emptyset, \emptyset\rangle$ and

$$\langle x, y, W, W, pid, \emptyset, \emptyset, \emptyset, \emptyset\rangle,$$

assuming pid is the thread period ID of t. $W_2(y)$ and $R_5(x)$ do not participate in generating 2v-blocks because $W_2(y) \notin FW(t)$ and $R_5(x) \notin IR(t)$.

For 2v-blocks $b = \langle v_1, v_2, op_1, op_2, pid, h_1, h_2, h_{12}, hmid_{12}\rangle$ and $b' = \langle v'_1, v'_2, op'_1, op'_2, pid', h'_1, h'_2, h'_{12}, hmid'_{12}\rangle$, where $(v_1 = v'_1 \wedge v_2 = v'_2) \vee (v_1 = v'_2 \wedge v_2 = v'_1)$, the operations op'_1 and op'_2 of b' can occur between the operations op_1 and op_2 of b if

$$\begin{aligned}&(h_{12} \cap hmid'_{12} = \emptyset)\\ &\wedge\ can\text{-}occur\text{-}between(\langle op'_1, h'_1, pid'\rangle, \langle op_1, op_2, h_{12}, pid\rangle)\\ &\wedge\ can\text{-}occur\text{-}between(\langle op'_2, h'_2, pid'\rangle, \langle op_1, op_2, h_{12}, pid\rangle).\end{aligned}$$

Two 2v-blocks b and b' are *atomic* with respect to each other, denoted $isAtomic2vBlk(b, b')$, iff the four operations cannot be interleaved to any of the unserializable patterns described above, according to the can-occur-between conditions described in the previous paragraph and Section 6.1.

To check atomicity of two transactions that share multiple variables, we have the following theorem, which extends Theorem 3:

Theorem 4. *Let t and t' be transactions with $thread(t) \neq thread(t')$. Suppose T does not have potential for deadlock. $\{t, t'\}$ is atomic iff*

1. $$\forall\, b \in 1v\text{-}blocks(t).\forall\, b' \in 1v\text{-}blocks(t').\ isAtomic1vBlk(b, b')$$

   and

2. $$\forall\, b \in 2v\text{-}blocks(t).\forall\, b' \in 2v\text{-}blocks(t').\ isAtomic2vBlk(b, b').$$

Proof. For the forward implication ($\Rightarrow$), we prove the contrapositive, which follows easily from the definitions of $isAtomic1vBlk$ and $isAtomic2vBlk$.

For the reverse implication ($\Leftarrow$), suppose all pairs of 1v-blocks and 2v-blocks are atomic. Let S be a nonserial trace for $\{t, t'\}$. We show that S is equivalent to some serial trace by the following three cases:

Case 1. Suppose there exists a variable x written by t and t'. Without loss of generality, we assume that t' performs the final write $e^{t'}_{FW(x)}$ to x in S. Let $e^{t}_{W(x)}$ denote a write to x in t. We can show that S is equivalent to the serial trace S' in which t precedes t', based on the following intermediate results, which can be proven based on the definitions of $isAtomic1vBlk$ and $isAtomic2vBlk$: 1) t cannot read any write of t' to any variable, 2) if t' reads some write of t in S, t' reads the same write in S', and 3) for each variable y accessed in both t and t', if there are writes to y in both t and t', $e^{t'}_{FW(y)}$ must occur after $e^{t}_{FW(y)}$ in S.

Case 2. Suppose no variable is written by both transactions and at least one transaction contains a write. Without loss of generality, suppose t contains a write e^{t}_{W}. If some read in t' reads the value written by e^{t}_{W}, then we can show that S is equivalent to the serial schedule in which t precedes t'; otherwise, we can show that S is equivalent to the serial schedule in which t' precedes t.

Case 3. If neither transaction contains a write, then S is trivially serializable. □

Let E and P be defined as in Section 6.1. The total number of 2v-blocks is $O(E^2)$. The algorithm based on Theorem 4 considers all pairs of 2v-blocks, so, assuming $|locksHeld(t)|$ is always bounded by a constant for all threads t, its worst-case running time is $O(PE^4)$.

6.3 Multiple Transactions that Share Multiple Variables

In the presence of multiple shared variables, a set T of transactions is not necessarily atomic even if all subsets of T with cardinality two are atomic. This is due to cyclic dependencies. For example, consider the following trace containing three transactions (time increases from left to right):

$$\begin{array}{lccccccc} t_1: & W(x) & & & & & W(y) & \\ t_2: & & R(x) & W(z) & & & & \\ t_3: & & & & R(z) & R(y). & & \end{array} \quad (6)$$

In any potential serial trace equivalent to this one, t_1 must precede t_2, t_2 must precede t_3, and t_3 must precede t_1. Due to the cyclic dependency, no equivalent serial trace exists. Therefore, $\{t_1, t_2, t_3\}$ is not atomic, even though all three subsets of T with cardinality two are atomic.

Cyclic dependencies between transactions that are pairwise atomic arise from dependencies involving initial reads and final writes. This observation motivates the following extension of Theorem 4: Let $IR\text{-}FW(T)$ denote the set of transactions obtained from T by discarding all events other than synchronization events and initial reads and final writes on shared variables.

Theorem 5. *Let T be a set of transactions. Suppose T does not have potential for deadlock. T is atomic iff for all t and $t' \in T$ with $thread(t) \neq thread(t')$,*

1. $$\forall\, b \in 1v\text{-}blocks(t).\forall\, b' \in 1v\text{-}blocks(t').\ isAtomic1vBlk(b, b')$$

   and

2. $\forall\, tr \in traces(IR\text{-}FW(T))$. *$tr$ is serializable.*

Proof. For the forward implication ($\Rightarrow$), the proof is straightforward. For the reverse implication ($\Leftarrow$), we need to prove that there is an equivalent serial trace S' for each trace S of all events in T. Condition 2 of this theorem implies Condition 2 of Theorem 4. Hence, for all t, $t' \in T$, $\{t, t'\}$ is atomic according to Theorem 4. Thus, only the sequence of initial reads and final writes of t and t' in S affects their possible order in S'. Condition 2 implies there is a serial trace S'' equivalent to $IR\text{-}FW(S)$. Therefore, S' can be obtained by concatenating the transactions in T in the same order that they appear in S''. □

This algorithm is relatively expensive because the number of possible traces may be large. On the positive side, this algorithm considers only traces for $IR\text{-}FW(T)$ and, hence, may be significantly faster than the naive algorithm that considers all traces for T.

The worst-case time complexity of the algorithm based on Theorem 5 is exponential in the number of events. This is not surprising because similar problems, such as determining serializability of a given trace, are NP-complete [25].

6.4 Usage and Comparison of the Three Block-Based Algorithms

The algorithms based on Theorems 3 and 4 can be applied to arbitrary executions. For Theorem 3, this simply means considering one shared variable at a time, i.e., applying the algorithm independently to each shared variable. For Theorem 4, this means considering the transactions pairwise and not checking atomicity of larger sets of transactions. Taking this perspective, we have three block-based algorithms that range from a relatively cheap one that detects limited but common kinds of atomicity violations to a relatively expensive one that also detects complex but rare kinds of atomicity violations. Based on the experiments in Section 8, we believe that, in practice, the algorithms based on Theorems 3 and 4 reflect better trade-offs between cost and defect-finding effectiveness. Indeed, in those experiments,

there is no atomicity violation that would be reported based on Theorem 5 and not reported based on Theorem 4 because the cyclic dependencies between three or more transactions that could cause such warnings never appear in those experiments.

6.5 Comparison of Reduction-Based Algorithm and Block-Based Algorithm

The block-based algorithm is more expensive than the reduction-based algorithm, but more accurate, according to the experimental results in Section 8. For a small example of this, consider the threads t_2, t_3, and t_4 in Fig. 4. Only x is shared, so the algorithm in Section 6.1 suffices. The 1v-blocks are $\langle x, R, dummy, false, false, pid_2, \{o_2\}, \{\}, \{\}\rangle$, $\langle x, R, dummy, false, false, pid_3, \{o_1\}, \{\}, \{\}\rangle$, and

$$\langle x, R, W, false, true, pid_4, \{\}, \{o_1, o_2\}, \{\}\rangle,$$

where pid_i is the ID of the thread period containing the events of transaction t_i. The block-based algorithm shows that $\{t_2, t_3, t_4\}$ is atomic. Recall from Section 5.4 that the reduction-based algorithm reports a false alarm for this example.

6.6 Dynamic Construction of Blocks

We construct 1v-blocks incrementally during execution of a transaction; this avoids storing all events in the transaction until its end.

2v-blocks are constructed when the transaction finishes; this requires storing only initial reads and final writes until the end of the transaction. As an optimization, if two initial reads e_1 and e_2 in a transaction operate on the same variable, $held(e_1) = held(e_2)$, and $heldmid(e_1, e_2) = \emptyset$, then one of them can be discarded without affecting the result.

To avoid constructing redundant blocks, the most recent several event patterns are cached. When an event pattern in the cache appears again, we do not construct a block for it again. This optimization saves times because constructing blocks is more expensive than a cache lookup.

The same block could appear in many transactions. We save space by sharing blocks among multiple transactions.

7 Instrumentation

This section describes the instrumentation of the source code.

We modify the pretty-printer in the Kopi [21] compiler to insert instrumentation as it pretty-prints the source code. The instrumentation intercepts the following events:

- reads and writes to all monitored fields (see below),
- entering and exiting synchronized blocks, including synchronized methods,
- entering and exiting methods that are considered as transactions (see below),
- calls to thread `start` and `join`,
- barrier synchronization.

The user specifies the classes to instrument as a list of expressions like `java.*`(denoting all classes in subpackages of `java`), `java.util.*`, or `java.util.Vector`.

By default, executions of the following code fragments in the instrumented classes are considered to be transactions: nonprivate methods, synchronized private methods, and synchronized blocks inside nonsynchronized private methods; as exceptions, the executions of the `main()` method in which the program starts and the executions of `run()` methods of classes that implement `Runnable` are not considered as transactions because these executions represent the entire executings of threads which are often not atomic. These defaults are taken from [10]. We include synchronized blocks here because locks are often used to achieve atomicity. We include nonprivate methods here because they are abstractions often expected by clients of the class to behave as atomic operations. The defaults can be overridden using a configuration file, e.g., the `run()` method of thread can be defined for atomicity checking. We did not override these defaults in any of the experiments in Section 8. In addition, start, join, and barrier operations are treated as transaction boundaries, as discussed in Section 4.

All nonfinal fields (with primitive type or reference type) of the specified classes are monitored. Accesses to these fields in all methods of all classes are instrumented because even methods not considered as transactions by themselves might be invoked during a transaction. Local variables are not monitored because they are necessarily thread-local. The defaults for monitoring nonfinal fields can also be overridden by a configuration file, e.g., some fields can be defined for nonmonitoring because they never escape.

In the reduction-based algorithm, for each monitored field, one or more locksets are maintained. In the block-based algorithm, for each monitored field, a previous event is cached to construct a block with the current event. We originally implemented these maps between monitored fields and their associated information as hash tables, with an object identifier combined with a field name as the key. This is relatively easy to implement but inefficient since each access requires a look up in the hash table. Our current implementation inserts in each monitored class a new field (call it `shadow_f`) corresponding to each monitored field `f` of the class. `shadow_f` points directly to the information associated with `f`.

There is no way to insert fields into array classes in Java, so we use the less efficient approach described above to associate shadow information with arrays, i.e., we maintain a hash table that maps each array reference a to shadow information a_s. Each array element has its own the shadow information. So, a_s is an array with the same dimension and size as a. When an array is created, its associated shadow array is created. As an optimization, parts of the array can be allocated dynamically when needed.

Monitoring every array element causes a large slowdown in some programs, so our system allows the user to specify a cutoff; for example, if the array is $[0..99] \times [0..99]$ and the cutoff is 3, then only the subarray $[0..2] \times [0..2]$ is monitored. Dynamic escape analysis is still carried out on the array and every element, regardless of the cutoff.

8 Experiments

We apply three algorithms to 12 programs. The three algorithms are: online reduction-based, offline reduction-based, and block-based.

```
TestCollections()
  for (i=0;i<numMethods;i++)
    for (j=0;j<numMethods;j++)
      if (methods i and j of C do not both take
          args of type C)
        TestTwoMethods(i,j);

TestTwoMethods(int i, int j)
  C o1 = new C(), o2 = new C();
  start a new thread that executes the ith method of
    o1 with o2 as a parameter if needed;
  start a new thread that executes the jth method of
    o2 with o1 as a parameter if needed;
```

Fig. 6. Pseudocode for test driver. `C` is `StringBuffer`, `Vector`, `Hashtable`, or `Stack`.

```
The reduction-based algorithms report:
  transaction Vector(Collection) is NOT atomic because :
    synchronized block @ Vector.java:689
    synchronized block @ Vector.java:266

The block-based algorithm reports:
  transaction Vector(Collection) is NOT atomic because :
  the unserializable pattern is
    VarName = Vector.elementCount
    Thread_1: R  @  Vector.java:267
    Thread_2: W  @  Vector.java:631
    Thread_1: R  @  Vector.java:690
```

Fig. 7. Excerpts of diagnostic information for the `Vector` example.

To facilitate comparison with [10], we use the same online reduction-based algorithm as in [10]; specifically, it uses the lockset algorithm from [29], ignores arrays, uses Theorem 1 instead of Theorem 2, and uses the improvements of Section 5.5. It does not use dynamic escape analysis or happen-before analysis.

The offline reduction-based algorithm is based on Theorem 2 and incorporates the multilockset algorithm (which uses dynamic escape analysis and happen-before analysis) for checking data races and the improvements of Section 5.5. Note that all these improvements could also be applied to the online reduction-based algorithm, but we did not do that in order to compare our algorithms with [10].

The block-based algorithm in these experiments is based on Theorem 4. Theorem 5 is more precise but more expensive than Theorem 4. In addition, we implemented a check for the presence of cyclic dependencies between three or more transactions and cyclic dependencies do not appear in these experiments. This implies that we did not miss any atomicity violations by using Theorem 4 instead of Theorem 5. We believe that Theorem 4 is sufficient for most programs.

The 12 programs are `elevator`, `tsp`, `sor`, and `hedc` from [29]; `moldyn`, `montecarlo`, and `raytracer` from [19]; `StringBuffer`, `Vector`, `Hashtable`, and `Stack` from Sun JDK 1.4.2; and `jigsaw` [20]. `elevator` simulates the actions of two elevators. `tsp` solves the traveling salesman problem; we run it on the accompanying data files `map4` and `map14`. `sor` is a scientific computing program which uses barriers rather than locks for synchronization. `hedc` is a Web crawler that searches astrophysics data on the Web. `moldyn`, `montecarlo`, and `raytracer` are computation-intensive parallel programs that compute molecular dynamics, Monte Carlo simulation, and ray tracing, respectively. `jigsaw` is a Web server implemented in Java. We instrument only its packages that are related with HTTP service. Table 1 shows the number of lines of code that are instrumented, i.e., it excludes code in uninstrumented libraries. For all programs that accept the number of threads as an argument, we use three threads. All experiments are done on a Sun Blade 1500 with a 1GHz UltraSPARC III CPU, 2GB RAM, SunOS 5.8, and JDK 1.4.2.

We modified `tsp` and `moldyn` slightly. Specifically, for `tsp`, we set `Tsp.MAX_NUM_TOURS` to 100 instead of 5,000 and used instances of `Object()` as lock objects instead of instances of `Integer(0)` since our system identifies locks by their identity hash code. For `moldyn`, we set `md.ITERS` to 1, moved some fields of `mdRunner` into its `run()` method so they became local variables, and marked instances of `particle` as unshared (i.e., accessed by only one thread) and, hence, did not record accesses to them; this annotation makes the analysis faster and does not affect the result.

We designed test drivers for the classes `StringBuffer`, `Vector`, `Hashtable`, and `Stack`. The pseudocode is shown in Fig. 6, where C denotes one of these classes. The driver creates two instances, o_1 and o_2, of C. For a pair $\langle m_1, m_2 \rangle$ of methods of C, the driver creates two threads, t_1 and t_2, where t_1 executes $o_1.m_1$ and t_2 executes $o_2.m_2$. Each execution of `TestTwoMethods` is analyzed separately for the atomicity checking. When a method requires an instance of C as argument, the other instance is used. For example, if C is `Vector`, and m_1 is `addAll`, then thread t_1 executes o_1.`addAll`(o_2). The driver tests all pairs of methods such that m_1 and m_2 do not both take an argument of type C; these excluded pairs would lead to potential for deadlock. For example, executing $\langle o_1.\texttt{addAll}(o_2), o_2.\texttt{removeAll}(o_1) \rangle$ may lead to deadlock because $o_1.\texttt{addAll}(o_2)$ locks o_1 then o_2, and $o_2.\texttt{removeAll}(o_1)$ locks o_2 then o_1. The driver does not test scenarios in which t_1 executes $o_1.m_1$ and t_2 executes $o_1.m_2$ because they would not produce any additional information since these methods are synchronized.

In these experiments, we check atomicity of transactions defined by the defaults in Section 7. For arrays, every element is monitored, except that we use a cutoff of 3 at the beginning of arrays for `moldyn`, `montecarlo`, and `raytracer` and we use a cutoff of 10 in the middle of arrays for `sor` to catch more violations.

8.1 Usability

The block-based algorithm provides more detailed diagnostic information than the reduction-based algorithms (online and offline). For example, Fig. 7 shows part of the output of these algorithms for the `Vector` example in Fig. 1. The reduction-based algorithms report an atomicity violation because of two consecutive synchronized blocks (e.g., $R...L...R...L$, which does not match the patterns in Theorems 1 and 2, see Fig. 5). The block-based algorithm reports an atomicity violation because it finds the second unserializable pattern described in Section 6.1, i.e., for the field `elementCount` of some instance of `Vector`, a write to that field by some thread (denoted `Thread_2`) executing line 631 of `Vector.java` can occur between two reads of the same field by another thread (denoted `Thread_1`)

TABLE 1
Performance and Accuracy

Program	lines of codes	Base time	On-line reduction		Off-line reduction		Block-based	
			time	report	time	report	time	report
elevator	528	0.2s	0.2s	0-2-0-0	0.5s	0-2-0	0.6s	0-2-0
tsp(map4)	706	0.24s	0.5s	0-0-1-0	0.5s	0-0-1	0.5s	0-0-0
tsp(map14)	706	0.46s	31.7s	0-1-0-1	32.5s	0-2-0	8m59s	0-2-0
sor	251	0.47s	2.2s	0-0-2-0	53.3s	0-0-0	1m4.1s	0-0-0
hedc	2197	0.6s	0.7s	0-0-2-0	1.0s	0-0-1	2.1s	0-0-0
moldyn	1265	44.03s	9m49s	0-0-0-2	38m22.1s	0-0-0	28m54.6s	0-0-0
montecarlo	3619	15.85s	1m43.3s	0-0-0-0	8m10.1s	0-0-0	8m11.4s	0-0-0
raytracer	1832	14.34s	35m13.6s	1-0-0-1	11m58.9s	2-0-0	36m17.6s	2-0-0
jigsaw	25012	1.60s	1.88s	1-2-1-1	2.74s	1-3-1	8m25.4s	1-3-0
StringBuffer	1255	-	-	0-1-0-0	-	0-1-0	-	0-1-0
Vector	1020	-	-	4-2-0-12	-	4-4-10	-	4-4-0
Hashtable	1054	-	-	0-4-1-0	-	0-4-1	-	0-4-0
Stack	119	-	-	3-2-0-14	-	3-4-12	-	3-4-0

The four categories of "report" for the online reduction algorithm are bug—benign—false alarm—missed violation. The three categories of "report" for the other two algorithms are bug—benign—false alarm. A dash for "time" means that the running time is negligible.

executing lines 267 and 690 of `Vector.java`. The reduction-based algorithms have an inherent limitation in reporting diagnostic information because they cannot indicate which variables are involved in the atomicity violation (variables involved in data races can be identified, but there is no data race in this example), while the block-based algorithm indicates the specific fields and accesses that violate atomicity.

8.2 Accuracy and Performance

Table 1 shows the running times and results of the three algorithms. "Base time" is the running time of the uninstrumented program. For each algorithm, "time" includes the running time of the instrumented program and the analysis. We classify warnings issued by each algorithm into three categories:

- *Bug*: The warning reflects a violation of atomicity that might cause a violation of an application-specific correctness requirement.
- *Benign*: The warning reflects a violation of atomicity that does not affect the correctness of the application.
- *False alarm*: The warning does not reflect a violation of atomicity.

Table 1 shows, for each category, the number of methods issued such that a warning in that category for a transaction that is an execution of that method or part code of that method. If a transaction is correctly reported as not atomic, the corresponding method is counted only under bug or benign, even if other warnings (which we do not need to classify) are also reported for that method. For a warning issued by the block-based algorithm, only the methods whose executions contribute two events in the unserializable patterns are counted. We aggregate the warnings in this way (instead of simply counting the number of warnings) to facilitate a fair comparison between the reduction-based and block-based algorithms. Note that the reduction-based algorithms always produce at most one warning per transaction (indicating that the patterns in Theorems 1 and 2 are not matched), while the block-based algorithm may produce multiple warnings per transaction since multiple parts of the transaction may match the unserializable patterns in Sections 6.1 and 6.2.

Table 1 also shows the number of missed errors for the online reduction algorithm, i.e., the number of atomicity violations that are reported by the offline reduction-based algorithm, but missed by the online reduction-based algorithm.

We conclude from Table 1 that:

1. The online reduction-based algorithm actually misses some atomicity violations in practice, for the reasons mentioned in Sections 5.4 and 5.5 and because it does not analyze arrays; this occurs for `tsp(map14)`, `moldyn`, `raytracer`, `jigsaw`, `Vector`, and `Stack`. For example, in `raytracer`, the online reduction-based algorithm misses an atomicity violation because it misclassifies some accesses to field `JGFRayTracerBench.checksum1` as race-free based on the information observed so far, whereas the offline algorithm classifies them as races based on the entire execution. Another example is in `moldyn`, the algorithms that analyze arrays (offline reduction-based and block-based) report atomicity violations involving arrays (these violations can be seen in Table 1), although these warnings are removed because of happen-before analysis (and, hence, are not evident in Table 3).
2. The block-based algorithm is more accurate than the online and offline reduction-based algorithms in the sense that it reports fewer false alarms.
3. For most programs, the offline reduction-based algorithm is slower than the online reduction-based algorithm because the latter is online (this avoids storage overhead), uses less accurate and faster data race analysis, and ignores array accesses. The median slowdown of the offline reduction-based algorithm compared to the online reduction-based algorithm is 46 percent. In `raytracer`, the offline reduction-based algorithm is faster because it uses escape analysis (this point is explained Section 8.3).
4. The block-based algorithm is slower than the offline reduction-based algorithm; the median slowdown is 20 percent. The block-based algorithm is relatively much slower for `tsp(map14)` and `jigsaw` because they execute a lot of code once (or a few times), producing many distinct blocks (see Table 4), while

the other programs iterate more, producing more duplicate blocks.

5. Different input data affects the runtime analysis result for some programs, such as `tsp(map4)` and `tsp(map14)`, where `tsp(map14)` exercises more code than `tsp(map4)`.

The bugs in `raytracer` come from atomicity violations involving the field `JGFRayTracerBench.checksum1`, which could get an incorrect value, causing the program to report failure. The bug in `jigsaw` is due to atomicity violations involving the field `w3c. tools. resources. store. ResourceStoreManager. loadedStore` due to statements `loadedStore++` and `loadedStore- -` without synchronization; as a result, `loadedStore` may contain an incorrect value. The error in `jigsaw` described in [30] does not appear in our experiments because the relevant code was modified in the newer version of `jigsaw` that we tested. The above atomicity violations involve data races. The errors in `Vector` and `Stack` are from atomicity violations on the field `elementCount` (as discussed in Section 1).

The offline reduction-based algorithm produces more false alarms than the block-based algorithm. For example, some `Collection` classes use `mod Count` to count modifications. Thus, when an update method m_1 executes `modCount++` (which is a read followed by a write) and another method m_2 checks for recent modifications by reading `modCount`, there is a serializable sequence of events $m_1{:}read(modCount)$ $m_2{:}read(modCount)$ $m_1{:}write(modCount)$. The block-based algorithm does not produce a warning. But, the benign data race on `modCount` may cause the reduction-based algorithms (online and offline) to produce an atomicity warning (a false alarm). Similar scenarios exist in `jigsaw` (e.g., on the field `alive` in the method `w3c.util.CachedThread.waitForRunner()`) and other programs.

For `Vector` and `Stack`, the online reduction-based algorithm produces fewer false alarms than the offline reduction-based algorithm because it misses some data races. By luck, the data races are benign and do not cause atomicity violations, but produce false alarms in the offline algorithm. The online algorithm uses [29]'s race detection algorithm, which assumes that the ownership of a variable is transferred when a second thread accesses the variable, but the ownership of a `Collection` class in our driver is not really transferred at that time. On the other hand, missed data races may cause the online reduction-based algorithm to miss some atomicity violations, as in `ray tracer`, discussed above.

8.3 The Benefits of Different Improvements

Table 2 shows the benefits of different improvements to the offline reduction-based algorithm. For each program, three groups of experimental data are shown: atomicity violations, data races, and execution time. The results for atomicity violations are aggregated as in Table 1, i.e., based on the method executed by the transaction. The results for data races are the numbers of fields for which warnings are issued. A field of a class is counted only once, even if warnings are issued for multiple instances of the class. The columns show cumulative improvements. For example, the column labeled with "happen-before" also uses escape analysis and the last column uses all four improvements.

In Table 2, "none" means that the lockset algorithm of [29] is used to detect data races. When "escape" or "happen-before" is used as an option, a revised Eraser lockset algorithm is used: When an object escapes, all its fields are regarded as in "exclusive" state; this corresponds to the state "exclusive2" in the lockset algorithm of [29]. With the option "happen-before," thread period IDs are used to track happen-before relations based on start-join and barriers. With the option "multilockset," the multilockset algorithm of Section 5.4 is used to detect data races. With the option "$\mathrm{AcqA}^*\mathrm{Rel}$," Theorem 2 is used instead of Theorem 1.

Table 3 compares the benefits of different improvements to the block-based algorithm. The columns show cumulative improvements. The "none" column means that only locks, not escape and happen-before information, are considered when determining whether an event can occur between two other events. For each improvement, the column "methods" reports the number of methods such that an atomicity warning is issued for a transaction which is an execution of that method or part code of that method and the column "fields" reports the number of fields such that an atomicity warning is issued involving accesses to that field. The three categories for "methods" are bug—benign—false alarm. The first category of "fields" is the numbers of fields such that an atomicity warning is issued for a 1v-block involving an access to that field; the second category of "fields" is the number of pairs of fields such that an atomicity warning is issued for a 2v-block involving accesses to these two fields.

We can see from Table 3 that dynamic escape analysis speeds up the block-based algorithm on several programs because it eliminates processing of accesses to unescaped variables (just checking whether they are escaped is fast). It also speeds up the offline reduction-based algorithms in several cases; this can be seen by comparing the first two columns in Table 2. Table 4 shows the ratio of unescaped events to total events on all variables. Besides improving efficiency, dynamic escape analysis can also eliminate some false alarms. For example, Table 2 shows that dynamic escape analysis removes many false alarms for data race and atomicity on `jigsaw`; Table 3 shows that several false alarms for atomicity are eliminated by dynamic escape analysis on `elevator`, `hedc`, and `Hashtable`.

Happen-before analysis can also eliminate some false alarms. For example, the happen-before analysis based on start and join removes false alarms on `tsp(map14)` in Table 2 and on `tsp(map4,map14)` and `hedc` in Table 3. The happen-before analysis based on barrier removes false alarms on `moldyn` in Table 2 and Table 3.

The multilockset algorithm eliminates some false alarms on `sor`; these false alarms remained even after escape and happen-before analysis were applied. The multilockset algorithm reveals data races missed by the revised Eraser lockset algorithm in `tsp(map14)` and `hedc`. This can be seen in Table 2.

Special treatment of $AcqA^*Rel$ (i.e., using Theorem 2 instead of Theorem 1) reduces the number of false alarms for some programs. For example, we can see in Table 2 that it eliminates some false alarms for atomicity in `hedc`, `jigsaw`, and `Hashtable`.

TABLE 2
The Benefits of Different Improvements to the Offline Reduction-Based Algorithm

Program		none	escape	happen-before	multi-lockset	AcqA*Rel
elevator	atmcty vlts	0-2-0	0-2-0	0-2-0	0-2-0	0-2-0
	data races	0-0-0-0	0-0-0-0	0-0-0-0	0-0-0-0	0-0-0
	time	0.4s	0.5s	0.5s	0.5s	0.5s
tsp(map4)	atmcty vlts	0-0-1	0-0-1	0-0-1	0-0-1	0-0-1
	data races	0-0-8-0	0-0-8-0	0-0-0-0	0-0-0-0	0-0-0
	time	0.5s	0.5s	0.5s	0.5s	0.5s
tsp(map14)	atmcty vlts	0-2-2	0-2-2	0-2-0	0-2-0	0-2-0
	data races	0-7-3-1	0-7-3-1	0-7-0-1	0-8-0-0	0-8-0
	time	1m1.3s	25.7s	32.6s	34.1s	32.5s
sor	atmcty vlts	0-0-2	0-0-2	0-0-2	0-0-0	0-0-0
	data races	0-0-2-0	0-0-2-0	0-0-2-0	0-0-0-0	0-0-0
	time	51.3s	51.0s	52.0s	53.4s	53.3s
hedc	atmcty vlts	0-0-3	0-0-3	0-0-3	0-0-3	0-0-1
	data races	0-1-0-1	0-1-0-1	0-1-0-1	0-2-0-0	0-2-0-0
	time	0.8s	1.0s	0.8s	1.0s	1.0s
moldyn	atmcty vlts	0-0-2	0-0-2	0-0-0	0-0-0	0-0-0
	data races	0-0-2-0	0-0-2-0	0-0-0-0	0-0-0-0	0-0-0
	time	42m30s	28m23.1s	34m11.7s	35m15.3s	38m22.1s
montecarlo	atmcty vlts	0-0-0	0-0-0	0-0-0	0-0-0	0-0-0
	data races	0-0-0-0	0-0-0-0	0-0-0-0	0-0-0-0	0-0-0
	time	12m40.5s	8m8.5s	8m16.5s	8m22.2s	8m10.1s
raytracer	atmcty vlts	out of	2-0-0	2-0-0	2-0-0	2-0-0
	data races	memory	1-0-0-0	1-0-0-0	1-0-0-0	1-0-0
	time	after 2 hours	12m1.2s	11m54.5s	11m50.8s	11m58.9s
jigsaw	atmcty vlts	1-3-3	1-3-2	1-3-2	1-3-2	1-3-1
	data races	2-8-28-0	2-12-0-0	2-12-0-0	2-12-0-0	2-12-0
	time	2.73s	2.69s	2.64s	2.80s	2.74s
StringBuffer	atmcty vlts	0-1-0	0-1-0	0-1-0	0-1-0	0-1-0
	data races	0-0-0-0	0-0-0-0	0-0-0-0	0-0-0-0	0-0-0
Vector	atmcty vlts	4-4-10	4-4-10	4-4-10	4-4-10	4-4-10
	data races	0-1-0-0	0-1-0-0	0-1-0-0	0-1-0-0	0-1-0
Hashtable	atmcty vlts	0-4-2	0-4-2	0-4-2	0-4-2	0-4-1
	data races	0-2-1-0	0-2-0-0	0-2-0-0	0-2-0-0	0-2-0
Stack	atmcty vlts	3-4-12	3-4-12	3-4-12	3-4-12	3-4-12
	data races	0-1-0-0	0-1-0-0	0-1-0-0	0-1-0-0	0-1-0

The three categories for atomicity violations are bug—benign—false alarm. The four categories for data races are bug—benign—false alarm—missed warning.

TABLE 3
The Benefits of Different Improvements to the Block-Based Algorithm

Program	None			Escape			happen-before		
	methods	fields	time	methods	fields	time	methods	fields	time
elevator	0-2-3	6-6	11.0s	0-2-0	2-0	0.6s	0-2-0	2-0	0.6s
tsp(map4)	0-0-1	1-0	0.5s	0-0-1	1-0	0.5s	0-0-0	0-0	0.5s
tsp(map14)	0-2-1	5-8	10m44.1s	0-2-1	5-8	9m24.3s	0-2-0	3-7	8m59s
sor	0-0-2	36-144	1m3.6s	0-0-2	36-144	1m3.7s	0-0-0	0-0	1m4.1s
hedc	0-0-5	5-2	9.8s	0-0-3	1-0	1.9s	0-0-0	0-0	2.1s
moldyn			>13hrs	0-0-2	6-12	29m8.4s	0-0-0	0-0	28m54.6s
montecarlo			>20hrs	0-0-0	0-0	8m18.9s	0-0-0	0-0	8m11.4s
raytracer			>20hrs	2-0-0	1-0	37m7.6s	2-0-0	1-0	36m17.6s
jigsaw			>20hrs	1-3-1	13-112	7m42.4s	1-3-0	13-102	8m25.4s
StringBuffer	0-1-0	1-2	-	0-1-0	1-0	-	0-1-0	1-0	-
Vector	4-4-0	3-0	-	4-4-0	3-0	-	4-4-0	3-0	-
Hashtable	0-4-1	2-0	-	0-4-0	2-0	-	0-4-0	1-0	-
Stack	3-4-0	4-0	-	3-4-0	4-0	-	3-4-0	4-0	-

A dash for "time" means that the running time is negligible. A blank for "methods" or "fields" means that the datum is unavailable.

The results for "fields" in the column "happen-before" of Table 3 shows that, for most programs, all of warnings are for 1v-blocks. This suggests that the algorithm in Section 6.1 is sufficient to find most atomicity violations. The algorithm in Section 6.2 is slower and the additional warnings it produces are typically more difficult to diagnose as bug or benign because they involve two variables and diagnosis requires understanding how updates to the two variables should be related.

8.4 Storage

Table 4 characterizes the storage used. Results for `Collection` classes are omitted because the storage used is small and depends mainly on the driver. "offline rdct storage" shows the storage of the offline reduction algorithm by the total size of all ($varsOne$ and $varsMul$) sets of variables in all transaction tree nodes discussed in Section 5.6. "# of blocks" shows the number of blocks (including 1v-blocks and 2v-blocks) stored by the block-based algorithm. "total

TABLE 4
Comparison of Storages and the Ratio between Unescaped Events and Escaped Events

Program	off-line rdct storage	# of blocks	total events	frac unesc events	multi-lkst size	Ersr lkst size
elevator	184	108	26K	0.43	86	20
tsp(map4)	60	155	1.3K	0.40	80	1
tsp(map14)	543	13474	14M	0.68	532	40
sor	64	3056	16M	0.97	49	0
hedc	350	1085	2.9K	0.21	386	31
moldyn	978	132	1.6G	0.86	2429	0
montecarlo	79	159	477M	0.99	92	0
raytracer	31	39	3.5G	0.99	29	1
jigsaw	2011	12508	8.4K	0.51	2169	110

events" is the total number of monitored events, including accesses to unescaped variables. "frac unesc events" is the ratio between the number of accesses to unescaped variables and the total number of events. "multilkst size" shows the sum of the maximum sizes of all sets (including $ReadSets$, $WriteSet$, $ReadThreadSet$, and $WriteThreadSet$ for each monitored variable) maintained by the multiple-lockset algorithm. "Ersr lkst size" shows the sum of the maximal sizes of all locksets maintained by Eraser [28] algorithms. The multilockset algorithm provides more accurate data race analysis with moderately increased storage. The relatively large difference on `moldyn` between "multilkst size" and "Ersr lkst size" is due to the use of barriers, which increase the number of thread period IDs. The Eraser lockset sizes are zero for `sor`, `moldyn`, and `montecarlo` because they use barrier synchronization, which is not monitored by the Eraser lockset algorithm.

8.5 Conclusions

In conclusion, the block-based algorithm is more accurate and produces more specific diagnostic information than the reduction-based algorithms. Their running times are often similar, although the block-based algorithm is much slower for some programs.

The experiments do not reveal any simple relationship between the running time and the number of events. This reflects the fact that the running time depends strongly on many other factors, e.g., how many events produce the same blocks, when variables escape, the number of thread periods, lockset sizes, etc.

Escape analysis improves both efficiency and precision (i.e., fewer false alarms). Happen-before analysis, the multilockset algorithm, and special treatment of $AcqA^*Rel$ also increase precision, but incur some overhead.

9 Related Work

In [31], we proposed the reduction-based and block-based algorithms for runtime atomicity checking. This paper describes several improvements to the algorithms and provides experimental results.

Flanagan and Freund [10] proposed a reduction-based algorithm with the improvements in Section 5.5. Their tool, called Atomizer, implements the online reduction-based algorithm described in Section 8.

Compared with Atomizer and our initial work [31], this paper contributes the following improvements to the reduction-based algorithms:

1. offline checking, which avoids missing atomicity violations due to miss-classification of events;
2. more accurate treatment of accesses to thread-local and read-only variables, as described in Theorem 2;
3. a new multilockset algorithm that produces fewer false alarms than previous lockset algorithms;
4. use of dynamic escape analysis, which reduces false alarms and often reduces running time;
5. use of happen-before analysis in data race detection to reduce false alarms;
6. on the implementation side, our system analyzes arrays; Atomizer does not.

Model checking can also be used to check atomicity [16], [8]. Model checking provides stronger guarantees than runtime monitoring because it explores all possible behaviors of a program. Also, many of the supporting analyses, such as dynamic escape analysis, analysis of array, deadlock detection, and special treatment of thread-local and read-only variables, etc., can be performed more easily and precisely in model checking than by program instrumentation [16]. However, model checking is more expensive and feasible only for programs with relatively small state spaces.

Flanagan et al. extended their atomicity type system to verify abstract atomicity of programs by analyzing purity [13]. We extended their work to verify atomicity of programs which use nonblocking synchronization [32].

Related work on runtime (also called *dynamic*) data race detection is discussed in Section 5.4. Choi et al. [6] combine static analysis and dynamic analysis and consider happen-before relations based on `start` and `join`. O'Callahan and Choi [24] extend the happen-before relation to consider `wait` and `notify` as well. Compared to the multilockset algorithm, [24] is more accurate but maintains more locksets. Our happen-before analysis ignores `wait` and `notify`, but considers barriers which [24] does not. Furthermore, we use dynamic escape analysis, whereas [6] uses static escape analysis. The reduction-based algorithm could be improved by using the race-detection techniques in [6], but it would still produce more false alarms than the block-based algorithm because imprecise race detection is only one of the causes of the additional false alarms.

Artho et al. developed a runtime analysis algorithm to detect *high-level data races* [2]. Absence of high-level data races is similar to atomicity. They introduce a concept of *view consistency* and utilize it to detect high-level data races. A *view* is the entire set of shared variables accessed in a synchronized block. Thread t_1 and thread t_2 are view consistent if the intersections of all views of t_1 with the maximal view of t_2 form a chain (with respect to the subset ordering $\subseteq$) and vice versa. View consistency and atomicity are incomparable (i.e., neither implies the other) [31].

Von Praun and Gross [30] present a static analysis to detect violations of *method consistency*, which is an extension of view consistency [2]. Method consistency and atomicity are also incomparable. Although their static analysis is unsound (in order to reduce the cost and the number of false alarms), it considers the entire program and therefore may be more thorough than runtime analysis in some cases. On the other hand, it produces more false alarms than our block-based algorithm, based on a comparison of the false alarms in our Table 1 with the false and spurious reports in Table 1 of [30].

Linearizability [18] is a correctness condition for objects which are shared by concurrent processes. Linearizability

can be viewed as a special case of strict serializability where transactions are restricted to consist of a single method applied to a single object [18]. Linearizability is defined semantically, i.e., in terms of the specification (correctness requirements) of the object. In contrast, we define atomicity in terms of operations performed by the implementation. Our definition is more restrictive, but has the practical benefit of being directly applicable to programs for which formal correctness requirements are unavailable.

ACKNOWLEDGMENTS

This work was supported in part by the US National Science Foundation under Grants CCR-0205376 and CNS-0509230 and by the US Office of Naval Research under Grants N00014-02-1-0363 and N00014-04-1-0722.

REFERENCES

[1] R. Agarwal, L. Wang, and S.D. Stoller, "Detecting Potential Deadlocks with Static Analysis and Runtime Monitoring," *Proc. Parallel and Distributed Systems: Testing and Debugging (PADTAD) Track of the 2005 IBM Verification Conf.*, Nov. 2005.

[2] C. Artho, K. Havelund, and A. Biere, "High-Level Data Races," *Proc. First Int'l Workshop Verification and Validation of Enterprise Information Systems (VVEIS)*, Apr. 2003.

[3] P.A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems.* Addison Wesley, 1987.

[4] C. Boyapati, R. Lee, and M. Rinard, "Ownership Types for Safe Programming: Preventing Data Races and Deadlocks," *Proc. 17th ACM Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pp. 211-230, Nov. 2002.

[5] C. Boyapati and M.C. Rinard, "A Parameterized Type System for Race-Free Java Programs," *Proc. 16th ACM Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, Nov. 2001.

[6] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan, "Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, pp. 258-269, 2002.

[7] M.B. Dwyer, J. Hatcliff Robby, and V.P. Ranganath, "Exploiting Object Escape and Locking Information in Partial-Order Reductions for Concurrent Object-Oriented Programs," *Formal Methods in System Design*, vol. 25, nos. 2-3, pp. 199-240, 2004.

[8] C. Flanagan, "Verifying Commit-Atomicity Using Model-Checking," *Proc. 11th Int'l SPIN Workshop Model Checking of Software*, 2004.

[9] C. Flanagan and S. Freund, "Type-Based Race Detection for Java," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, pp. 219-232, 2000.

[10] C. Flanagan and S.N. Freund, "Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs," *Proc. ACM Symp. Principles of Programming Languages (POPL)*, pp. 256-267, 2004.

[11] C. Flanagan and S.N. Freund, "Type Inference against Races," *Static Analysis Symp. (SAS)*, Aug. 2004.

[12] C. Flanagan, S.N. Freund, and M. Lifshin, "Type Inference for Atomicity," *Proc. ACM SIGPLAN Int'l Workshop Types in Languages Design and Implementation (TLDI)*, Jan. 2005.

[13] C. Flanagan, S.N. Freund, and S. Qadeer, "Exploiting Purity for Atomicity," *Proc. ACM Int'l Symp. Software Testing and Analysis (ISSTA)*, pp. 221-231, 2004.

[14] C. Flanagan and S. Qadeer, "A Type and Effect System for Atomicity," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, 2003.

[15] C. Flanagan and S. Qadeer, "Types for Atomicity," *Proc. ACM SIGPLAN Int'l Workshop Types in Languages Design and Implementation (TLDI)*, pp. 1-12, 2003.

[16] J. Hatcliff Robby and M.B. Dwyer, "Verifying Atomicity Specifications for Concurrent Object-Oriented Software Using Model-Checking," *Proc. Fifth Int'l Conf. Verification, Model Checking and Abstract Interpretation (VMCAI)*, Jan. 2004.

[17] K. Havelund, "Using Runtime Analysis to Guide Model Checking of Java Programs," *Proc. Seventh Int'l SPIN Workshop Model Checking of Software*, Aug. 2000.

[18] M.P. Herlihy and J.M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Trans. Programming Languages and Systems*, vol. 12, no. 3, pp. 463-492, July 1990.

[19] Java Grande forum, Java Grande Multithreaded Benchmark Suite, version 1.0, http://www.javagrande.org/, 2001.

[20] Jigsaw, version 2.2.4, http://www.w3c.org, 2004.

[21] Decision Management Systems GmbH, Kopi compiler, http://www.dms.at/kopi/, 2002.

[22] R.J. Lipton, "Reduction: A Method of Proving Properties of Parallel Programs," *Comm. ACM*, vol. 18, no. 12, pp. 717-721, 1975.

[23] F. Mattern, "Virtual Time and Global States of Distributed Systems," *Proc. Int'l Workshop Parallel and Distributed Algorithms*, pp. 120-131, 1989.

[24] R. O'Callahan and J.-D. Choi, "Hybrid Dynamic Data Race Detection," *Proc. ACM SIGPLAN 2003 Symp. Principles and Practice of Parallel Programming (PPoPP)*, pp. 167-178, 2003.

[25] C.H. Papadimitriou, "The Serializability of Concurrent Database Updates," *J. ACM*, vol. 26, no. 4, pp. 631-653, Oct. 1979.

[26] W. Pugh, "Fixing the Java Memory Model," *Proc. ACM Conf. Java Grande*, pp. 89-98, 1999.

[27] E. Ruf, "Effective Synchronization Removal for Java," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, pp. 208-218, June 2000.

[28] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T.E. Anderson, "Eraser: A Dynamic Data Race Detector for Multithreaded Programs," *ACM Trans. Computer Systems*, vol. 15, no. 4, pp. 391-411, Nov. 1997.

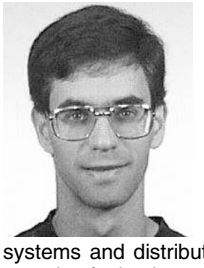
[29] C. von Praun and T.R. Gross, "Object Race Detection," *Proc. 16th ACM Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, Oct. 2001.

[30] C. von Praun and T.R. Gross, "Static Detection of Atomicity Violations in Object-Oriented Programs," *J. Object Technology*, vol. 3, no. 6, June 2004.

[31] L. Wang and S.D. Stoller, "Run-Time Analysis for Atomicity," *Proc. Third Workshop Runtime Verification (RV '03)*, 2003.

[32] L. Wang and S.D. Stoller, "Static Analysis for Programs with Non-Blocking Synchronization," *Proc. ACM SIGPLAN 2005 Symp. Principles and Practice of Parallel Programming (PPoPP)*, June 2005.

Liqiang Wang is a PhD candidate in the Computer Science Department at the State University of New York at Stony Brook. His research focuses on improving the reliability of software, specifically, designing, and implementing tools for effective testing, analysis, and verification of programs.



Scott D. Stoller received the Bachelor's degree in physics, summa cum laude, from Princeton University in 1990 and the PhD degree in computer science from Cornell University in 1997. He is an associate professor in the Computer Science Department of the State University of New York at Stony Brook. His primary research interest is methods and tools for design, analysis, testing, and verification of software, especially software for concurrent systems and distributed systems, including specialized techniques for ensuring fault tolerance and security. His research also includes work on program optimization and incremental computation. He received a US National Science Foundation CAREER Award in 1999 and a US Office of Naval Research Young Investigator Award in 2002.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.