Runtime Analysis of Atomicity for Multi-threaded Programs

Liqiang Wang Scott D. Stoller
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400
{liqiang,stoller}@cs.sunysb.edu

Abstract

Atomicity is a semantic 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 multi-threaded programs, an interface usually contains several procedures (or methods), whose invocations 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 checks atomicity by efficiently analyzing permutations of the order of events in a trace that are consistent with the synchronization. To improve the efficiency and accuracy of both algorithms, we incorporate a multi-lockset algorithm for checking data races, dynamic escape analysis, and start-join analysis. Experiments show that both algorithms are effective in finding atomicity violations.

Index Terms: concurrent programming, testing and debugging, Java, data race, atomicity.

1 Introduction

Multi-threading has become a common programming technique. Not only operating systems but also many applications are multi-threaded. However, developing multi-threaded programs is difficult, because concurrency introduces the possibility of errors that do not exist in sequential programs. Multi-threaded 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 multi-threaded 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 deadlock and data race. A *deadlock* occurs when all threads are blocked, each waiting for some action by one of the other threads. A *data race* occurs when two threads concurrently access a shared variable and at least one of the accesses is a write.

Numerous static and runtime analysis techniques are designed to ensure that concurrent programs are

```
public class Vector extends ... implements ... {
    public Vector(Collection c) {
 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[]) { ... }

    public synchronized void removeAllElements() { ... }
    public synchronized boolean add(Object o) { ... }
}

Thread_1                          Thread_2
 Vector v2 = newVector(v1);         v1.removeAllElements();
                                    // v1.add(o);
```

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

free of deadlocks and data races [FF00, BR01, BLR02, SBN+97, CLL+02]. 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 Figure 1. Consider the following execution of the program at the bottom of Figure 1: `thread_1` constructs a new vector `v2` from another vector `v1` with k elements and then yields execution to `thread_2` immediately after statement 1, `thread_2` removes all elements of `v1`, and then `thread_1` resumes execution at statement 2. The incorrect outcome (based on the behavior of `toArray`) is that `v2` has k elements, all of which are `null`. Another 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`. Again, based on the behavior of `toArray`, `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, like transactions, 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 Figure 1 can be considered to have two transactions, corresponding to invocations of `Vector(Collection)` and `removeAllElements()`, and is, obviously, not equivalent to any serial execution. Therefore, these methods violate atomicity. Similarly, the second scenario also shows a violation of atomicity.

Flanagan *et al.* developed a type system for atomicity [FQ03a, FQ03b]. It can ensure that methods are atomic in all possible executions. However, the type system often requires manual annotation of the program, and is sometimes undesirably restrictive (*i.e.*, the type checker rejects some well-behaved programs as well as all badly-behaved programs).

This paper presents two runtime algorithms for detecting potential violations of atomicity: reduction-

based algorithm and block-based algorithm. Runtime analysis is less powerful than type-based approach, because it cannot ensure correctness of the system in unexplored paths, but may be more precise (*i.e.*, give fewer false alarms) for the explored path. Furthermore, runtime analysis is automatic, which 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 a violation is possible in other executions (in the same explored path) because of the nondeterminism of thread scheduling. We implemented both algorithms. Experiments show that they can successfully find subtle errors.

The reduction-based algorithm is an extension of our original reduction-based algorithm [WS03] and Flanagan and Freund's Atomizer algorithm [FF04]. It first determines how locks are used to protect shared variables and then uses this information to infer commutativity properties of events. If the sequence of events in a transaction matches a given commutativity pattern, then the transaction is atomic.

The block-based algorithm [WS03] determines whether atomicity is violated in an observed trace or any permutation of the trace that is consistent with the synchronization events in the trace. This is checked efficiently by considering interactions of pairs of events (called *blocks*) from different transactions.

Our system instruments the source code by inserting code that sends events to the monitor. The monitor implements both detection algorithms and can apply them *on-line* (*i.e.*, during execution of the program) or *off-line* (*i.e.*, after the program terminates).

One direction for future work is to decrease the overhead by using static analysis, as in [CLL+02], to show absence of data races or atomicity violations in parts of the program, and applying 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 describe the reduction-based algorithm and block-based algorithm, respectively. Sections 5 and 6 present dynamic escape analysis and start-join analysis, respectively, which can be used to improve both algorithms. Section 7 describes instrumentation of the source code. Section 8 contains experimental results. Related work is discussed in Section 9.

2 Background

This section reviews the standard notion of serializability [BHG87] and then introduces our notion of atomicity.

An event e is an instance of one of the operations: $R(x)$, which reads variable x; $W(x)$, which writes variable x; $acq(l)$, which acquires a lock l; or $rel(l)$, which releases a lock l. For example, `synchronized(l)` {*body*} in Java indicates two events (in addition to the events performed by the body): $acq(l)$ at the entry point and $rel(l)$ at the exit point. For a read or write event e, 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 events *conflict* if they act on the same variable and at least one event 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. We do not consider nested transactions: if method a calls method b, we consider the invocation of method a as a transaction that includes all of the events in the execution of b.

A *trace* tr is a sequence of events from a set of transactions, which may come from different threads. Let $trans(tr)$ denote the set of transactions which form tr.

Given a set T of transactions, a *trace* for T is an interleaving of the transactions in T that is consistent with the original order of events from each thread and with the synchronization events. For example, an $acq(l)$ event in one transaction cannot appear between an $acq(l)$ and a $rel(l)$ in another transaction. In this paper, we enforce consistency with respect to only acquire and release operations on locks, and start and join operations on threads. 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. A read without a write-predecessor in tr is called an *uninitialized read* in tr.

Two traces tr_1 and tr_2 are *equivalent* iff (*i*) they are merges of the same set of transactions, (*ii*) each read event has the same write-predecessor in both traces, and (*iii*) each variable has the same final write event in both traces. This corresponds to view equivalence in transaction processing [BHG87].

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.

3 Reduction-based Algorithm

In this section, we present an atomicity checking algorithm that is based on Lipton's reduction theorem [Lip75]. The idea is to infer atomicity from commutativity properties of events.

3.1 Commutativity Properties

Following [Lip75, FQ03b], 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 without changing the resulting state. An event is a *left-mover* if, whenever it appears immediately after an event of a different thread, the two events can be swapped without changing the resulting state.

For example, if an event e_1 of thread t_1 is a lock acquire in a trace, its immediate successive event e_2 from another thread can not 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 conservative approximate classification of events can conveniently be obtained based on synchronization operations. An access to a variable is *race-free* if it is not involved in any data race, as defined in Section 1.

Theorem 3.1. *Lock acquire events are right-movers. Lock release events are left-movers. Race-free reads and race-free writes are both-movers.*

Proof. A proof sketch follows; a more detailed proof appears in [FQ03b]. Commutativity of acquire and release events is discussed above. Race-free reads and race-free writes are both-movers, because race-freedom implies that an immediately following or immediately preceding event by another thread cannot be conflicting access to the same variable, so swapping the events does not affect the result. □

3.2 Reduction-Based Algorithm

Given an arbitrary interleaving of events in a set T of transactions, if all events of each transaction can be moved together (by repeatedly swapping adjacent events in the trace) without changing the results of reads and without changing the final writes, 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, and each event in t that precedes e can be moved to the right (towards e), and each event in t that follows e can be moved to the left (towards e), then all events of each transaction can be moved together.

A trace that ends in deadlock with some thread in the middle of a transaction is not equivalent to any serial trace, so we require that all traces for T have no potential for deadlock. Following [Hav00], we say that transactions t and t' have a *potential for deadlock* if they acquire two locks ℓ_1 and ℓ_2 in different orders without first acquiring some other lock that prevents their attempts to acquire ℓ_1 and ℓ_2 from being interleaved. This can be checked with the goodlock algorithm [Hav00]. This approach is approximate because it considers only pairs of threads.

These observations motivate the following theorem.

Theorem 3.2. *A set T of transactions is atomic if T has no potential for deadlock, and each transaction in T has the form $R^*N^?L^*$, where R, L, and N denote right-mover, left-mover, and non-mover, respectively.*

Proof. This is a simple variant of Lipton's reduction theorem [Lip75]. □

This theorem, together with a technique for detecting data races (such the lockset algorithm in [SBN+97]), leads directly to an efficient and conservative algorithm for checking atomicity. But this algorithm reports false alarms in many cases. The following sections show how to improve it.

3.3 Improvement 1: Read-only and Thread-local Variables

If a variable is accessed by a single thread (*i.e.*, *thread-local*), or there are only read accesses on it (*i.e.*, *read-only*), obviously there is no data race on the variable, therefore all accesses on it are both-movers.

Consider a sequence of events starting with an acquire, ending with the matching release, and containing only accesses to thread-local and read-only variables. Such a sequence matches the pattern RB^*L. A transaction containing multiple such sequences does not match the pattern in Theorem 3.2 but may be atomic. For example, if x is read-only or thread-local, the following set of two transactions is atomic, even though the hypothesis of Theorem 3.2 is not satisfied.

$$\begin{array}{l} acq(\ell_1)\ R(x)\ rel(\ell_1)\ acq(\ell_2)\ R(y)\ rel(\ell_2) \\ acq(\ell_1)\ R(y)\ rel(\ell_1)\ acq(\ell_2)\ R(x)\ rel(\ell_2) \end{array} \tag{1}$$

We extend Theorem 3.2 to show that such sets of transactions are atomic. We do this in two steps.

Lemma 3.3. *Given a set T of transactions, T is atomic if T has no potential for deadlock and each transaction in T has the form $(R + AcqRel)^*N^?(L + AcqRel)^*$, where $AcqRel$ denotes an acquire of some lock immediately followed by a release of the same lock.*

Proof. Based on Theorem 3.2, it suffices to argue that $AcqRel$ can be ignored when determining atomicity. It can be ignored because it has no effect on the state of the program and it has no effect on the commutativity

properties of other operations (e.g., it does not affect whether any accesses to variables are race-free). 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. □

Theorem 3.4. *A set T of transactions is atomic if T has no potential for deadlock and each transaction in T has the form $(R + AcqA^*Rel)^*N^?(L + AcqA^*Rel)^*$, where R, L, and N denote right-mover, left-mover, and non-mover respectively, and $AcqA^*Rel$ denotes an acquire of some lock, followed by accesses to read-only or thread-local variables, then followed by release of the same lock.*

Proof. This follows from Lemma 3.3 and the fact that events in A commute with all events from other threads, so they have no effect on atomicity. □

On-line classification of accesses as read-only or thread-local is based on whether the variable has been read-only or thread-local so far. Off-line classification is based on the entire execution and is therefore more accurate.

3.4 Improvement 2: Multi-Lockset Algorithm for Checking Data Race

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. Data races can be detected statically or dynamically. This paper focuses on dynamic detection.

The lockset algorithm in [SBN+97] is 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. [SBN+97] supposes that the initialization period ends when the variable is accessed by a second thread; this is an unsafe approximation (*i.e.*, it may miss 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, a warning is issued, indicating a potential data race. To see why this treatment of initialization is unsafe, if the thread that allocates x accesses x after x escapes and before a second thread accesses x, and no lock is held at the accesses by the first and second threads, then a data race occurs, but this algorithm does not report it.

Praun and Gross [vPG01] modify the lockset algorithm by introducing a more sophisticated condition for determining when initialization ends. It supposes 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, [vPG01] 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.

[FF04] improves 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 read and write to the variable, and the *write-protecting* lockset contains locks held on every write to the 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 data race warning is reported.

We propose the *multi-lockset algorithm*, which is more accurate than the preceding algorithms. It incurs higher overhead but is still practical, according to the experiments in Section 8. The main three improvements are: (1) Our algorithm uses a dynamic escape (from thread) analysis described in Section 5 to determine when "initialization" of a variable ends, *i.e.*, when to start refining the variable's lockset. This improves precision because only the accesses before escaping are ignored. The dynamic escape analysis is conservative, so our algorithm does not miss any races. (2) The happens-before relation based on `start` and `join` operations on threads is considered. If a thread t starts another thread t', the events in t before t' is started cannot be concurrent with the events in t'. Thus, there is no need to hold the same lock at both accesses. Similarly, if a thread t calls `join` on thread t', then events in t after that call cannot be concurrent with events in t'. Our algorithm to analyze the start-join relation is described in Section 6. (3) Multiple-reader, single-writer scenarios are analyzed more accurately by maintaining a set of locksets for each variable.

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, $WriteSet(x) := WriteSet(x) \cap locksHeld(t)$.

$ReadSets(x)$ and $WriteSet(x)$ are not updated by accesses to x before x escapes. Let $t_1(x)$ denote the first thread that accesses x after x escapes, or $\perp$ if there is no such thread. If no thread is concurrent with $t_1(x)$ (our technique to determine whether two threads are concurrent is described in Section 5), then accesses to x by $t_1(x)$ are also ignored when computing $ReadSets(x)$ and $WriteSet(x)$.

When the program terminates, if x never escapes, or is accessed by only one thread after escaping, there is no race on it. Otherwise, there are three cases: (1) $WriteSet(x)$ is not initialized; this means that there is no write to x, so there is no data race on x. (2) $WriteSet(x)$ is empty; this means that all writes to x do not have a common lock, so there is a potential data race. (3) $WriteSet(x)$ is not empty; in this case, each lockset in $ReadSets(x)$ is intersected with $WriteSet(x)$. If all these intersections are not empty, there is no data race on x, otherwise, a potential data race is reported.

This algorithm is practical because *ReadSets* usually contains only a few sets, according to the experiments in Section 8.

This algorithm is more accurate than previous lockset-based algorithms. For example, suppose x has escaped from its creating thread, and the threads in Figure 2 execute in the order t_3, t_2, t_1 (ignore t_1' for now). According to the definition of data race, there is no data race on x. The algorithms in [SBN+97, vPG01, FF04, CLL+02] all report a false alarm (potential data race on x). The multi-lockset algorithm does not.

Even the multi-lockset algorithm produces some false alarms. For example, consider the threads t_1', t_2 and t_3 in Figure 2 (ignore t_1 now). If the execution order is t_3, t_2, t_1', since $locksHeld(t_1'.read(x)) \cap locksHeld(t_1'.write(x)) = \emptyset$, a potential data race is reported, but this is a false alarm, and it causes a false alarm in atomicity checking. In Section 4, we will see that the block-based algorithm does not produce a false alarm for this example.

```
t1:                  t2:                  t3:                  t1':
synchronized(o1){    synchronized(o2){    synchronized(o1){    read(x);
 synchronized(o2){     read(x);             read(x);           synchronized(o1,o2){
   write(x);         }                    }                     write(x);
   write(x);                                                   }
 }
}
```

Figure 2: An example to illustrate the accuracy of the multi-lockset algorithm.

3.5 Other Improvements

The classification of all lock acquires and releases as right-movers and left-movers, respectively, in Section 3.1 can be refined. In the following cases, they are as both-movers [FF04, HRD04].

- *Re-entrant 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.
- *Thread-local locks.* If a lock is used by only one thread, acquire and release on it are both-movers.
- *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).

Off-line algorithms can classify thread-local locks and protected locks more accurately than on-line 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 on-line algorithms. This may cause atomicity violations to be missed.

3.6 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 implementing the sets in sorted order, but we did not implement this because most locksets are small.

We instrument the program by a source-to-source transformation. The instrumented program constructs and stores a tree structure for each transaction during execution. Each node other than the root corresponds to a synchronized block (*i.e.*, an execution of a synchronized statement or synchronized method), and is labelled with the acquired lock. The tree structure reflects the nesting of synchronized blocks. Each node is also labelled with the set of variables accessed only once (ignoring accesses in sub-blocks) in the corresponding synchronized block, and with the set of variables accessed multiple times. This tree compactly represents all of the necessary information about the transaction, because Theorem 3.4 implies that we only need to distinguish the multiplicities "?" and "*".

4 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 events. We design algorithms for three different cases: multiple transactions that share exactly one variable; two transactions that share multiple variables; and multiple transactions that share multiple variables. Locks are not counted as shared variables.

4.1 Multiple Transactions That Share Exactly One Variable

Given a set T of transactions, the algorithm looks for unserializable patterns of events of T. An *unserializable pattern* is a sequence in which events 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 from 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 events in the patterns are on the same variable (the exactly one shared variable). These patterns can be drawn as follows, where each line corresponds to a transaction, and time advances from left to right.

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

T is atomic if no feasible interleaving of events of T matches any of these patterns (this idea is formalized in Theorem 4.2 below).

The block-based algorithm looks for these unserializable patterns by considering pairs of "blocks" from different transactions. We introduce the idea of blocks because: many events in a transaction are identical from the perspective of atomicity (*e.g.*, they operate on the same variable, the same locks are held etc.). Combining events into blocks eliminates this kind of redundancy automatically. Informally, a block is a pair of read or write events from one transaction, together with information about synchronization. Specifically, for two events e_1 and e_2 in transaction t with $var(e_1) = var(e_2)$, call the variable v, there is a block for e_1 and e_2 if 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 uninitialized read of v in t.

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 *block* for e_1 and e_2 is a tuple $\langle op(e_1), op(e_2), fw(e_1), fw(e_2), held(e_1), held(e_2), held(e_1, e_2)\rangle$, where $op(e)$ is the operation, namely, $R(v)$, $W(v)$ or dummy; $fw(e)$ is a boolean value indicating whether e is the final write on v in t; $held(e)$ is the set of locks held by the thread when executing event e; and $held(e_1, e_2)$ is the set of locks held continuously from e_1 to e_2.

For example, the transaction

$$t : acq(\ell_1)\ R(v)\ acq(\ell_2)\ W(v)\ R(v)\ rel(\ell_2)\ rel(\ell_1) \tag{2}$$

has two blocks,

$$\begin{array}{l}\langle R(v),\ W(v), \text{false}, \text{true}, \{\ell_1\}, \{\ell_1, \ell_2\}, \{\ell_1\}\rangle \\ \langle W(v), R(v), \text{true}, \text{false}, \{\ell_1, \ell_2\}, \{\ell_1, \ell_2\}, \{\ell_1, \ell_2\}\rangle.\end{array} \tag{3}$$

The information about held locks is used to determine feasibility of interleavings of events from different blocks. For example, to determine whether an event e of a block can occur between events e_1 and e_2 of another block, we check whether $held(e) \cap held(e_1, e_2)$ is empty. This simple test is accurate provided there is no potential for deadlock in the program. So we check potential for deadlock (as described in Section 3.2) as part of the block-based algorithm. To see that this test may be inaccurate if there is potential for deadlock, note that e cannot occur between e_1 and e_2 in the following example, even though $held(e) \cap held(e_1, e_2) = \emptyset$.

$$\begin{array}{l}t : acq(l_1)\ acq(l_2)\ rel(l_2)\ e\ rel(l_1) \\ t' : acq(l_2)\ acq(l_1)\ rel(l_1)\ e_1\ e_2\ rel(l_2)\end{array} \tag{4}$$

Note that many pairs of events may produce the same block, and only one copy of the block is stored. This can greatly reduce the space needed by the algorithm.

Two blocks b and b' for transactions of different threads are *atomic*, denoted $isAtomicBlk(b, b')$, if the synchronization indicated by the locksets in the blocks prevents the unserializable patterns described above, *i.e.*, no three out of the four events in the two blocks can form one of those patterns.

Let $blocks(t)$ denote the set of blocks for a transaction t. To check atomicity of multiple transactions which share exactly one variable, we have the following theorem.

Lemma 4.1. *Let t and t' be transactions that share exactly one variable, with $thread(t) \neq thread(t')$. $\{t, t'\}$ is atomic iff $\forall\ b \in blocks(t)$ and $\forall\ b' \in blocks(t')$, $isAtomicBlk(b, b')$ holds.*

Proof. See appendix. □

Theorem 4.2. *Let T be a set of transactions that share exactly one variable. T is atomic iff $\forall\ t, t' \in T$, if $thread(t) \neq thread(t')$, then $\forall\ b \in blocks(t)$ and $\forall\ b' \in blocks(t')$, $isAtomicBlk(b, b')$ holds.*

Proof. See appendix. □

Let E be the total number of events in all transactions of T. The number of blocks is $O(E)$, because rule A for constructing blocks combines each event with at most one preceding event, and rule B constructs at most $O(E)$ blocks because there is only one final write for each variable in each transaction. Assuming $|locksHeld(t)|$ is always bounded by a constant for all threads t, the cost of checking $held(e) \cap held(e_1, e_2) = \emptyset$ is $O(1)$, and the worst-case running time of the algorithm based on Theorem 4.2 is $O(E^2)$.

4.2 Two Transactions That Share Multiple Variables

To check atomicity of two transactions that share multiple variables, the test embodied in Theorem 4.2 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 unserializability. If they all operate on the same variable, the analysis in Section 4.1 applies. Suppose they operate on two variables. If they contain no conflicting events, or exactly one pair of conflicting events, they do not cause unserializability. Suppose they contain two pairs of conflicting events. We can check based on the definition of serializability in Section 2 whether every feasible interleaving of these four events is serializable; if so, the two blocks are atomic. A few illustrative cases of unserializable interleavings are listed in the following.

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 $FW(t)$ be the set of final writes on shared variables in t. Let $UR(t)$ denote the set of uninitialized reads on shared variables in t. Let $heldmid(e_1, e_2)$ be the locks acquired and released after e_1 and before e_2, and not included in $held(e_1) \cup held(e_2)$. To check whether events e_1' and e_2' from transaction t' can occur between e_1 and e_2 from transaction t, we need to check $held(e_1, e_2) \cap heldmid(e_1', e_2') = \emptyset$, $held(e_1, e_2) \cap held(e_1') = \emptyset$, and $held(e_1, e_2) \cap held(e_2') = \emptyset$.

A *2-block* for a transaction t is a tuple $\langle op(e_1), op(e_2), held(e_1), held(e_2), held(e_1, e_2), heldmid(e_1, e_2)\rangle$ 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 $FW(t) \cup UR(t)$. Let *2-blocks(t)* denote the set of *2-blocks* for transaction t. For example, for the following transaction t, $UR(t) = \{R_1(x)\}$, $FW(t) = \{W_3(x), W_4(y)\}$, and *2-blocks(t)* contains $\langle R_1(x), W_4(y), \emptyset, \emptyset, \emptyset, \emptyset\rangle$ and $\langle W_3(x), W_4(y), \emptyset, \emptyset, \emptyset, \emptyset\rangle$.

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

Two 2-blocks b and b' are *atomic*, denoted $isAtomic2Blk(b, b')$, if the synchronization indicated by the sets of locks in the blocks prevents the unserializable patterns described above.

To check atomicity of two transactions that share multiple variables, we have the following theorem.

Theorem 4.3. *Let t and t' be transactions with $thread(t) \neq thread(t')$. $\{t, t'\}$ is atomic iff (i) $\forall\, b \in blocks(t)$, $\forall\, b' \in blocks(t')$, $isAtomicBlk(b, b')$ holds and (ii) $\forall\, b \in$ 2-blocks(t), $\forall\, b' \in$ 2-blocks(t'), $isAtomic2Blk(b, b')$ holds.*

Proof. See appendix. □

Let E be the total number of events in all transactions of T. Assuming $|locksHeld(t)|$ is always bounded by a constant for all threads t, the worst-case running time of the algorithm based on Theorem 4.3 is $O(E^4)$.

4.3 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):

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

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 arise only from conflicts involving uninitialized reads and final writes. This motivates the following theorem. Let *UR-FW*(T) denote the set of transactions obtained from T by discarding all events other than synchronization events and uninitialized reads and final writes on shared variables.

Theorem 4.4. *Let T be a set of transactions. T is atomic iff $\forall\ t, t' \in T$, if thread(t) $\neq$ thread(t'), then (i) $\forall\ b \in$ blocks(t) and $\forall\ b' \in$ blocks(t'), isAtomicBlk(b, b') holds; (ii) $\forall\ b \in$ 2-blocks(t) and $\forall\ b' \in$ 2-blocks(t'), isAtomic2Blk(b, b') holds; and (iii) $\forall\ tr \in$ traces(UR-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. For all $t, t' \in T$, because $\{t, t'\}$ is atomic, only the sequence of uninitialized reads and final writes of t and t' in S affects their possible order in S'. Condition (iii) implies there is a serial trace S'' equivalent to *UR-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 expensive, because the number of possible traces may be large. On the positive side, this algorithm considers only traces for *UR-FW*(T), and hence may be significantly faster than the naive algorithm that considers all traces for T. According to our experiments in Section 8, such cyclic dependencies are rare, so executing the algorithms in Section 4.1 and 4.2 should be sufficient for most programs.

The block-based algorithm based on Theorem 4.4 works in exponential complexity in the worst case. This is not surprising, because similar problems, such as determining serializability of a given trace, are NP-complete [Pap79].

4.4 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_1', t_2 and t_3 in Figure 2. Only x is shared, so the algorithm in Section 4.1 applies. The blocks are $\langle R(x), W(x), false, true, \{\}, \{o_1, o_2\}, \{\}\rangle$, $\langle R(x), dummy, false, false, \{o_2\}, \{\}, \{\}\rangle$, and $\langle R(x), dummy, false, false, \{o_1\}, \{\}, \{\}\rangle$. The block-based algorithm shows that $\{t_1', t_2, t_3\}$ is atomic. Recall from Section 3.4 that the reduction-based algorithm reports a false alarm for this example.

4.5 Improvements to Block-based Algorithm

4.5.1 Escape Analysis and Start-join Analysis

Escape analysis and start-join analysis are easy to incorporate in the block-based algorithm. An access to a variable that has not yet escaped is not used to form blocks. When determining feasibility of an interleaving of events, ordering constraints from start-join analysis (as well as locking) are considered.

4.5.2 Dynamic Construction of Blocks

The simplest block-based algorithm constructs all blocks for a transaction when the transaction finishes. This requires storing all events in the transaction until the end of the transaction. This could be expensive for long transactions. We avoid this by constructing blocks incrementally during the execution of the transaction.

2-blocks are constructed when the transaction finishes, but this requires storing only uninitialized reads and final writes. If two uninitialized reads e_1 and e_2 in a transaction operate on the same variable, and $held(e_1) = held(e_2)$, and $heldmid(e_1, e_2) = \emptyset$, then one of them can be discarded without affecting the result.

To avoid constructing redundant blocks, the recent several events patterns to construct blocks are cached, when the same events pattern in cache appears again, we do not need to construct these kind of blocks. This optimization saves times. It does not change space usage, because the redundancy would be eliminated anyway when inserting the redundant block into the set of blocks.

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

5 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 instance 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.
- o is an instance of `Thread` (including its subclasses) and `o.start` is called. Note that, if o is 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.

To indicate whether an object has escaped, a boolean field `escaped` is added into every instrumented class. Its initial value is false. Reflection is used to find all objects to which a given object refers. If some methods (*e.g.*, native methods or methods in library classes) are not instrumented, then escape information can be maintained conservatively by assuming that all arguments to uninstrumented methods escape. For some widely used library classes (in particular, collections and maps), we use hand-written wrappers that track escape information accurately without instrumenting the source code of the library classes. Instrumenting library code is sometimes complicated by dependencies between classes, so we instrument library classes only in experiments that specifically test atomicity of library classes.

6 Start-Join Analysis

This section describes our algorithm for tracking happens-before relationships induced by `start` and `join`. This information is used as described in Sections 3.4 and 4.5.1. Happens-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.

We assign an identification number (ID) to each thread. Each event is labelled with the ID of the executing thread. When a thread t_1 calls t_2`.start()` to start another thread t_2 with ID id_2, we change the ID of t_1 from id_1 to id_1'. Events labelled with id_1 cannot be concurrent with events labelled with id_2. When a thread t_1 calls t_2`.join()` to wait for thread t_2 with ID id_2 to terminate, the ID of t_1 is changed from id_1' to id_1''. Events labelled with id_1'' cannot be concurrent with events labelled with id_2. A tree with an edge for each thread ID is used to store the temporal ordering relations between thread IDs. Each node of the tree is labelled with `start` or `join`.

A more efficient but more complicated alternative is to use vector clocks [OC03].

7 Instrumentation

This section describes the instrumentation of the source code.

We modify the pretty-printer in the Kopi [kop] 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.
- entering and exiting methods that are considered as transactions (see below).
- calls to thread `start` and `join`.

The user specifies the classes to instrument as a list of expressions like `java.*` (denoting all classes in sub-packages 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: public methods, protected methods, synchronized private methods, and synchronized blocks inside non-synchronized private methods; as exceptions, the `main()` method in which the execution of the program starts and `run()` methods of classes that implement `Runnable` are not considered as transactions. This is based in part on the defaults in [FF04]. The defaults can be overridden using a configuration file.

All non-final 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 non-final fields can also be overridden by a configuration file.

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. 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`.

In our system, each array element is treated as distinct variable which also has the shadow information. There is no way to insert fields into array classes in Java, so for arrays we use a different implementation of the above map. For each field `f` with array type, we insert a new "array shadow" field `array_shadow_f` with array type (in addition to `shadow_f` for the array reference) with the same dimension and size as the original array. Each element of `array_shadow_f` points to the shadow information for the corresponding element of f. Similarly, for each local variable v with array type in a method m, because the array might escape from the local scope, we create a new local variable `array_shadow_v`. Each assignment to a field or local variable with array type is augmented with an assignment to the corresponding array shadow of the field or local variable, respectively. Our implementation currently does not handle methods that return arrays. Monitoring every array element causes 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.

8 Experiments

We apply three algorithms to 12 programs. The three algorithms are: on-line reduction-based, off-line reduction-based, and block-based (based on Theorem 4.3). The off-line reduction-based algorithm is based on Theorem 3.4, and incorporates with the multi-lockset algorithm for checking data races, dynamic escape analysis and start-join analysis. To facilitate comparison with [FF04], we use the same on-line reduction-based algorithm as in [FF04]; specifically, it uses the lockset algorithm from [vPG01], ignores arrays, and uses Theorem 3.2 instead of Theorem 3.4.

The 12 programs are `elevator`, `tsp`, `sor`, and `hedc` from [vPG01]; `moldyn`, `montecarlo`, and `raytracer` from [JGB]; `StringBuffer`, `Vector`, `Hashtable`, and `Stack` from Sun JDK 1.4.2; and `jigsaw` [jig]. `elevator` simulates the actions of two elevators. `tsp` solves the travelling salesman problem; we run it on the accompanying data file `map4`. `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. `hedc` and `sor` use Doug Lea's concurrent programming library [Lea]. `moldyn`, `montecarlo`, and `raytracer` are parallel computation-intensive programs. We design test drivers for the classes `StringBuffer`, `Vector`, `Hashtable`, and `Stack`. For each of these classes, say C, 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 $o_2.m_2$. The driver tests all pairs of methods such that m_1 and m_2 do not both take an argument of type C; the excluded pairs would lead to potential for deadlock. When a method requires an instance of C as argument, the other instance is used. 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. `jigsaw` is a Web server implemented in Java. We instrument only its packages which are related with HTTP service. Table 2 shows the number of lines of code that are instrumented (*i.e.*, it does not include code in uninstrumented libraries). For all programs that accept the number of threads as an argument, we use three threads by default. 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.

By default, we check atomicity on transactions defined by the defaults in Section 7. Accesses to all non-final fields are monitored. For arrays, every element is monitored, except that we use a cutoff of 3 for

```
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
```

Figure 3: Excerpts of diagnostic information for the `Vector` example.

`sor`, `moldyn`, and `raytracer`.

8.1 Usability

The block-based algorithm provides more detailed diagnostic information than the reduction-based algorithms (on-line and off-line). For example, Figure 3 shows part of the output of these algorithms for the `Vector` example in Figure 1. Note that the reduction-based algorithms report an atomicity violation because of two consecutive synchronized blocks (*i.e.*, RB*LB*RB*L), but 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 that the `elementCount` field is involved, and points out the specific 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 transactions which issue warnings in that category are reported. For the block-based algorithm, only the transaction that contributes two events in the unserializable pattern is counted. All of the algorithms may produce multiple warnings for each atomicity-violating transaction; typically the block-based algorithm produces more warnings.

Table 1 also shows the number of missed errors for the on-line reduction algorithm, *i.e.*, the number of atomicity-violating transactions for which no warning is produced.

Program	Base	On-line reduction		Off-line reduction		Block-based	
	time	time	report	time	report	time	report
elevator	0.2s	0.2s	0-2-0-0	0.5s	0-2-0	0.7s	0-2-0
tsp	0.24s	0.5s	0-1-0-0	2.9s	0-1-0	2.5s	0-1-0
sor	0.47s	2.2s	0-2-0-0	1m12.1s	0-2-0	1m10.6s	0-2-0
hedc	0.6s	0.7s	1-0-0-0	1.1s	1-0-0	2.4s	1-0-0
moldyn	44.03s	7m22.5s	0-0-0-3	31m31.5s	0-3-0	32m43.1s	0-3-0
montecarlo	15.85s	1m43.3s	0-0-0-0	8m1.8s	0-0-0	8m2.6s	0-0-0
raytracer	14.34s	35m13.6s	1-0-0-1	11m20s	2-0-0	11m16.4s	2-0-0
jigsaw	1.60s	1.75s	1-1-0-2	2.84s	1-3-1	172m44s	1-3-0
StringBuffer	-	-	1-0-0-0	-	1-0-0	-	1-0-0
Vector	-	-	6-0-0-2	-	6-2-10	-	6-2-0
Hashtable	-	-	2-0-3-0	-	2-0-3	-	2-0-0
Stack	-	-	5-0-0-2	-	5-2-12	-	5-2-0

Table 1: Performance and Accuracy. The four categories of "report" for the on-line 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.

We conclude from Table 1 that: (1) The on-line reduction-based algorithm actually misses some atomicity violations in practice, for the reasons mentioned in Sections 3.4 and 3.5; this occurs for `moldyn`, `raytracer`, `jigsaw`, `Vector` and `Stack`, because the on-line data race detection algorithm mis-classifies some accesses as race-free. (2) The block-based algorithm is more accurate than the on-line or off-line reduction-based algorithms, in the sense that it reports fewer false alarms. (3) For most programs, the running time of the off-line reduction-based algorithm is within a factor of 5 of the running time of the on-line reduction-based algorithm; the latter is faster because it is on-line (avoiding storage overhead), uses less accurate and cheaper data race analysis, and ignores array accesses. (4) For all programs except for `jigsaw`, the running times of the two off-line algorithms (reduction-based and block-based) are similar.

The error in `hedc` is because of atomicity violations on `PooledExecutorWithInvalidate.poolSize_` which stores the number of live threads. Before a thread terminates, it checks `poolSize_` to make sure that there are still live threads to execute the remaining tasks. But if all threads perform this check before updating `poolSize_`, and then all threads terminate, there may be undone tasks. The errors in `raytracer` come from atomicity violations on `JGFRayTracerBench.checksum1`, which could get an incorrect value, causing the program to report failure. The error in `jigsaw` is from atomicity violations on the field `w3c.tools.resources.store.ResourceStoreManager.loadedStore` due to statements `loadedStore++` and `loadedStore--`; as a result, `loadedStore` may contain an incorrect value. The error in `jigsaw` described in [vPG03] does not appear, 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 `StringBuffer`, `Vector`, `Hashtable`, and `Stack` are from atomicity violations on the fields `elementCount` (discussed in Section 1), `Count`, and `modCount` (discussed below).

Besides improving accuracy, dynamic escape analysis makes the algorithms that use it (off-line reduction-based and block-based) faster than the algorithm that does not (on-line reduction-based) on `raytracer`, because for an access to an unescaped variable, the former algorithms merely process events on the escaped fields, which the latter algorithm processes events on both unescaped and escaped fields. Table 2 shows the

ratio of accesses to unescaped *vs.* escaped variables. The block-based algorithm is much slower than the reduction-based algorithm on `jigsaw`. We believe this is because in our test scenario, `jigsaw` executes few iterations of loops; in contrast, while the other programs execute more iterations, and many of the resulting events produce the same block.

Start-join analysis eliminates some false alarms. For example, it shows that the events in the `main` method of `sor` (the `main` method is treated as a transaction in this program) cannot be concurrent with the events in the threads that `main` starts.

Array accesses do affect atomicity in some of these programs. For example, in `moldyn`, the algorithms that analyze arrays (off-line reduction-based and block-based) report benign atomicity violations involving arrays.

Using Theorem 3.4 instead of Theorem 3.2 (*i.e.*, special treatment of $AcqA^*Rel$) reduces the number of false alarms for some programs. For example, the on-line reduction-based algorithm produces a false alarm for `Hashtable.equals(Collections o)` because it finds multiple sequential synchronized blocks. These synchronized blocks match the $AcqA^*Rel$ structure.

The off-line reduction-based algorithm produces more false alarms than the block-based algorithm. For example, some `Collection` classes use `modCount` 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 pattern $m_1 : read(modCount)$ $m_2 : read(modCount)$ $m_1 : write(modCount)$. The block-based algorithm does not report a violation. But the data race on `modCount` may cause the reduction-based algorithms (on-line and off-line) to produce a warning. Similar scenarios exist in `jigsaw` (*e.g.*, the field `alive` in the method `w3c.util.CachedThread.waitForRunner()`) and other programs.

For `Vector` and `Stack`, the on-line reduction-based algorithm produces fewer false alarms than the off-line reduction-based algorithm, because it misses some data races. In the on-line reduction-based algorithm, the ownership of a variable is transferred when a second thread accesses the variable. Recall that our driver for `Collection` classes uses two concurrent threads. The on-line reduction-based algorithm misses some data races because it wrongly transfers the ownerships of some variables between the two threads. These data races cause more false alarms in the off-line reduction-based algorithm for the reason described in the previous paragraph. On the other hand, missed data races may cause the on-line reduction-based algorithm to miss some atomicity violations.

In our experiments with the block-based algorithm, most atomicity violations can be found by the algorithm in Section 4.1 (applied once for each shared variable), which is significantly faster than the algorithm in Section 4.2. Also, the additional warnings produced by the algorithm in Section 4.2 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.

8.3 Storage

Table 2 characterizes the storage used. Results for `Collection` classes are omitted, because the storage is small and depends mainly on the driver. "max locks held" is the maximum number of locks held at once by a thread. "off-line reduction storage" shows the storage by the total number of entries in the data structures of the off-line reduction-based algorithm, where each entry mainly stores the name of a variable. "block storage" shows the number of blocks (including 2-blocks) stored by the block-based algorithm. "ratio of unesc/esc

program	lines of code	max locks held	off-line reduction storage	block storage	ratio of unesc/esc events	multi lockset size	Eraser lockset size
`elevator`	528	1	177	166	0.25	44	20
`tsp`	706	1	94	516	0.008	51	8
`sor`	251	0	98	528	0	6	4
`hedc`	2197	3	258	1131	0.28	62	30
`moldyn`	1265	0	1413944	454	1.82	0	0
`montecarlo`	3619	0	52	159	1394	0	0
`raytracer`	1832	1	14	39	7760	2	0
`jigsaw`	25012	6	836	49954	1.17	520	301

Table 2: Comparison of storages, and the ratio between unescaped events and escaped events.

events" is the ratio between the numbers of unescaped events and escaped events. "multi-lockset size" and "Eraser lockset size" show the sum, when the program terminates, of the sizes of all locksets maintained by the multiple-lockset and Eraser [SBN+97] algorithms, respectively. The multi-lockset algorithm provides more accurate data race analysis with moderately increased storage, roughly, a factor of two for most of these programs.

9 Related Work

In [WS03], we proposed the reduction-based and block-based algorithms to dynamically check atomicity of programs. This paper improves the accuracy and efficiency of the algorithms and provides experimental results.

Flanagan and Freund [FF04] proposed a reduction-based algorithm with the improvements in Section 3.5. Their tool, called Atomizer, implements the on-line reduction-based algorithm described in Section 8.

Compared with Atomizer and our initial work [WS03], this paper contributes the following improvements to the reduction-based algorithms: (1) off-line 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 3.4; (3) a new multi-lockset algorithm that produces fewer false alarms than previous lockset algorithms; (4) use of dynamic escape analysis instead of unsafe heuristics; (5) use of start-join 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 [HRD04, Fla04]. Model checking provides stronger guarantees than runtime monitoring algorithms, because it explores all possible behaviors of a program, but model checking is feasible only for programs with relatively small state spaces.

Related work on runtime (also called *dynamic*) data race detection is discussed in Section 3.4. [CLL+02] combines static analysis and dynamic analysis and considers happen-before relation based on `start` and `join`. [OC03] extends the happens-before relation to consider `wait` and `notify` as well. Our multi-lockset algorithm is more precise than the lockset algorithm they use. Furthermore, we use dynamic escape analysis, whereas [CLL+02] uses an approximate static escape analysis.

Artho *et al.* developed a runtime analysis algorithm for detecting *high-level data races* [AHB03]. Absence

of high-level data races is similar to atomicity. They introduce a concept of *view consistency* which is utilized 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 $V(t_1)$ with the maximal view of $V(t_2)$ 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) [WS03].

Praun and Gross [vPG03] present a static analysis to detect violations of *method consistency*, which is an extension of view consistency [AHB03]. Method consistency and atomicity are also incomparable. Although their 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, but it produces more false alarms (based on a comparison of the false alarms in our Table 1 with the false and spurious reports in Table 1 of [vPG03]).

Linearizability [HW90] 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. Linearizability is defined more semantically than atomicity: linearizability is defined in terms of the specification (correctness requirements) of the object, while atomicity is defined in terms of operations performed by the implementation. The latter is more restrictive but has the practical benefit of being directly applicable to programs for which formal correctness requirements are unavailable.

References

[AHB03] Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. In *Proc. First International Workshop on Verification and Validation of Enterprise Information Systems (VVEIS)*, April 2003.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison Wesley, 1987.

[BLR02] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proc. 17th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 211–230, November 2002.

[BR01] Chandrasekar Boyapati and Martin C. Rinard. A parameterized type system for race-free Java programs. In *Proc. 16th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, volume 36(11) of *SIGPLAN Notices*, pages 56–69. ACM, November 2001.

[CLL+02] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 258–269. ACM, 2002.

[FF00] Cormac Flanagan and Stephen Freund. Type-based race detection for Java. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 219–232. ACM Press, 2000.

[FF04] Cormac Flanagan and Stephen N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proc. of ACM Symposium on Principles of Programming Languages (PLDI)*. ACM Press, 2004.

[Fla04] Cormac Flanagan. Verifying commit-atomicity using model-checking. In *SPIN 2004*, volume 2989 of *Lecture Notes in Computer Science*, pages 252–266. Springer-Verlag, 2004.

[FQ03a] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 2003.

[FQ03b] Cormac Flanagan and Shaz Qadeer. Types for atomicity. In *Proc. ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI)*, pages 1–12. ACM Press, 2003.

[Hav00] Klaus Havelund. Using runtime analysis to guide model checking of Java programs. In *Proc. 7th Int'l. SPIN Workshop on Model Checking of Software*, volume 1885 of *Lecture Notes in Computer Science*. Springer-Verlag, August 2000.

[HRD04] John Hatcliff, Robby, and Matthew B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In *Proceedings of the Fifth International Conference on Verification, Model Checking and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*. Springer-Verlag, January 2004.

[HW90] Maurice P. Herlihy and Jeanette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[JGB] Java Grande Forum, Java Grande Benchmark Suite, thread version 1.0. Available from http://www.javagrande.org/.

[jig] Jigsaw, version 2.2.4. Available from http://www.w3c.org.

[kop] DMS Decision Management Systems GmbH, Kopi compiler. Available from http://www.dms.at/kopi/.

[Lea] Doug Lea. Package util.concurrent. Available from http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html.

[Lip75] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.

[OC03] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *ACM SIGPLAN 2003 Symposium on Principles and Practice of Parallel Programming*, pages 167–178. ACM, 2003.

[Pap79] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.

[vPG01] Christoph von Praun and Thomas R. Gross. Object race detection. In *Proc. 16th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, volume 36(11) of *SIGPLAN Notices*, pages 70–82. ACM Press, October 2001.

[vPG03] Christoph von Praun and Thomas R. Gross. Static detection of atomicity violations in object-oriented programs. In *Workshop on Formal Techniques for Java-like Programs(FTfJP)*, July 2003.

[WS03] Liqiang Wang and Scott D. Stoller. Run-time analysis for atomicity. In *Third Workshop on Runtime Verification (RV03)*, volume 89(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2003.

Appendix

Accesses to unshared variables do not affect atomicity of a trace, so we restrict accesses to shared variables in these proofs.

Lemma 4.1: Let t and t' be transactions that share exactly one variable, with $thread(t) \neq thread(t')$. $\{t, t'\}$ is atomic iff $\forall\ b \in blocks(t)$ and $\forall\ b' \in blocks(t')$, $isAtomicBlk(b, b')$ holds.

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

For the reverse implication ($\Leftarrow$), suppose $isAtomicBlk(b, b')$ holds for all pairs of blocks b and b'. Let S be a non-serial 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:

(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 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 $isAtomicBlk(b, b')$ would be false for some block b' containing $e^{t'}_R$, a contradiction.

(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 non-atomic pair of blocks would exist), and that S is equivalent to the serial trace in which t precedes t'. □

Theorem 4.2: Let T be a set of transactions that share exactly one variable. T is atomic iff $\forall\ t, t' \in T$, if $thread(t) \neq thread(t')$, then $\forall\ b \in blocks(t)$ and $\forall\ b' \in blocks(t')$, $isAtomicBlk(b, b')$ holds.

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 non-serial trace for T. For $T' \subseteq T$, let $S|T'$ denote the subsequence of S containing only events from transactions in T' (otherwise an unserializable pattern would be formed). Let t be the transaction that performs the final write e^t_{FW} in S. Let T_2 be the set of transactions 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. For all $t_1 \in T_1$ with $thread(t) \neq thread(t_1)$, the hypothesis of the contrapositive case and

Lemma 4.1 imply that $\{t, t_1\}$; since t_1 does not read t's write, and t performs the final write in S, $t1$ 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 $S_1 \cdot t \cdot S_2$. □

Theorem 4.3 Let t and t' be transactions with $thread(t) \neq thread(t')$. $\{t, t'\}$ is atomic iff (i) $\forall$ $b \in blocks(t)$, $\forall$ $b' \in blocks(t')$, $isAtomicBlk(b, b')$ holds and (ii) $\forall$ $b \in$ 2-blocks(t), $\forall$ $b' \in$ 2-blocks(t'), $isAtomic2Blk(b, b')$ holds.

Proof. For the forward implication ($\Rightarrow$), we prove the contrapositive, which follows easily from the definitions of *isAtomicBlk* and *isAtomic2Blk*.

For the reverse implication ($\Leftarrow$), suppose all pairs of blocks and 2-blocks are atomic. Let S be a non-serial 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 proved based on the definitions of *isAtomicBlk* and *isAtomic2Blk*: (i) t cannot read any write of t' to any variable; (ii) if t' reads some write of t in S, t' reads the same write in S'; (iii) 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. □