CS2K707(P) Seminar Report

on

Atomicity Analysis for Multi-threaded Programs

Submitted In Partial Fulfilment Of The Degree Of

Bachelor Of Technology

by

Pallav Gupta
Y1.232, S7 CSE

Department of Computer Science & Engineering
National Institute of Technology, Calicut
2004 Monsoon

National Institute of Technology, Calicut
Department of Computer Science & Engineering

Certified that this Seminar Report entitled

Atomicity Analysis for Multi-threaded Programs

is a bonafide report of the Seminar presented by

Pallav Gupta
Y1.232, S7 CSE

in partial fulfilment of the degree of
Bachelor of Technology

Mr.Vinod P.
Seminar Coordinator
Lecturer
Dept.of Computer Science & Engineering

Dr.V.K.Govindan
Professor and
Head
Dept.of Computer Science & Engineering

Abstract

Ensuring the correctness of multi-threaded programs is difficult, due to the potential for unexpected and nondeterministic interactions between threads. In the past, researchers have addressed this problem by devising tools for detecting race conditions, a situation where two threads simultaneously access the same data variable, and at least one of the accesses is a write. However, verifying the absence of such simultaneous-access race conditions is neither necessary nor sufficient to ensure the absence of errors due to unexpected thread interactions.

A stronger non-interference property, namely atomicity, is required. A method is atomic if its execution is not affected by and does not interfere with concurrently executing threads. Precisely, a method (or in general a code block) is atomic if for every (arbitrarily interleaved) program execution, there is an equivalent execution with the same overall behavior where the atomic method is executed serially, that is, the method's execution is not interleaved with actions of other threads.

This report discusses two methods of atomicity checking ,namely static and dynamic checking. Type system is for specifying and verifying the atomicity of methods in multi-threaded programs. Dynamic analysis is used to find out atomicity violations in multi-threaded softwares. For run-time analysis two algorithms reduction-based algorithm and block-based algorithm are discussed.

Contents

1 Introduction **1**
1.1 Type Systems for Checking Atomicity Violation 1
1.2 Dynamic Analysis for Multi-threaded Programs 1

2 The Need for Atomicity **2**

3 Static Type System **4**
3.1 Theory of LEFT and RIGHT Movers 4
3.2 Verification of Atomicity in Multi-threaded Programs : An Example 4

4 Basic Atomicities **6**

5 Runtime Analysis of Atomicity for Multi-threaded Programs **7**
5.1 Atomicity in Transactions 7
5.2 Reduction-Based Algorithm 9
5.2.1 Commutativity Properties 9
5.2.2 Reduction-Based Algorithm 9
5.2.3 Read-only and Thread-local Variables 10
5.2.4 Multi-Lockset Algorithm for Checking Data Race 11
5.2.5 Other Improvements 12
5.3 Implementation of Reduction-based Algorithm 12

6 Block-based Algorithm **13**
6.1 Multiple Transactions That Share Exactly One Variable 13
6.2 Two Transactions That Share Multiple Variables 14
6.3 Multiple Transactions That Share Multiple Variables 15

7 Comparison of Reduction-based Algorithm and Block-based Algorithm **16**

8 Instrumentation **17**

9 Conclusion **18**

1 Introduction

This paper presents static and dynamic analysis for detecting atomicity violations in multi-threaded software. This analysis combines ideas from both Lipton's theory of reduction and earlier dynamic race detectors. This report describes type systems for checking atomicity violations in multi-threaded software. This report describes two algorithms for run time detection of atomicity violations and compares their cost and effectiveness.

1.1 Type Systems for Checking Atomicity Violation

A type system is implemented for specifying and checking atomicity properties of methods in multi-threaded programs. Methods can be annotated with the keyword *atomic*. Type system checks that for any (arbitrarily-interleaved) execution, there is a corresponding serial execution with equivalent behavior in which the instructions of the atomic method are not interleaved with instructions from other threads.

1.2 Dynamic Analysis for Multi-threaded Programs

The reduction-based algorithm checks atomicity based on commutativity properties of events in a trace. The block-based algorithm checks atomicity by efficently analyzing permutations of the order of events in a trace that are consistent with synchronization. Experiments show that both algoritms are efficient in finding atomicity violations.

The reduction-based algorithm 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 transaction is atomic. The block-based algorithm determines whether atomicity is violated in an observed trace or any permuatation 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.

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

Runtime analysis is less powerful than type-based approach,because it can not ensure correctness of the system in unexplored path, but may be more precise.(i.e. give fewer false alarms) for explored path. Furthermore, runtime analysis is automatic, which is significant practical advantage.

2 The Need for Atomicity

As an illustration of the problems that arise in multi-threaded programming, consider the java program shown below. This program allocates a new bank account, and makes two deposits into the account in parallel. This program is written in the language Concurrent Java, which is essentially a multi-threaded subset of Java extended with *let* and *fork* constructs. This program

```
Class Account {
int balance = 0;          //since this value is zero
int deposit1(int x) {
this.balance = this.balance + x;
}
}
let Account a = new Account in {     //object created a
fork {a.deposit1(10)};
fork {a.deposit1(10)}
}
```

Figure 1: Java Code

may exhibit unexpected behavior. In particular, if the two calls to *deposit1* are interleaved, the final value of balance may reflect only one of the two deposits made to the account, which is clearly not the intended behavior of the program. That is, the program contains a race condition: two threads attempt to manipulate the field *deposit1* simultaneously, with incorrect results.

This error can be fixed by protecting the field *balance* by the implicit lock of the account object and only accessing or updating *balance* when that lock is held.

```
int deposit2(int x) {//second version of deposit funcation
Synchronized (this) {
this.balance = this.balance + x;
}
}
```

Figure 2: Improved Code with Lock Mechanism

In general, however, the absence of race conditions does not imply the absence of errors due to thread interactions. To illustrate this point, extend the account implementation with two additional methods *readBalance1* to return the current account balance and *withdraw1* to take money out of the account.

readbalance1

```
int    readBalance1   { int t;
synchronize(this){t=balance;};
return t;}
```

withdraw1

```
void  withdraw1 (int amt) {
int    b=readBalance1();
synchronize(this){
balance=b-amt;}}
```

Even though there are no races in either method, the method *withdraw1* is not atomic and may not behave correctly. For example, we consider two concurrent transactions on the account a withdrawal and a deposit, issued at a time when the account balance is 10.

fork withdraw1(10); ; // Thread 1

fork deposit2(10); ; // Thread 2

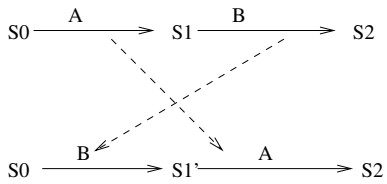
We would expect an account balance of 10 after the program terminates, but certain executions violate this expectation. Suppose the scheduler first performs the call to *readBalance1* in Thread 1 which returns 10. The scheduler then switches to Thread 2 and completes the execution of *deposit2* ending with balance = 20. Finally, the scheduler switches back to Thread 1 and completes the execution setting balance to 0. Thus, even though there are no races in this program, unexpected interaction between the threads can lead to incorrect behavior.

3 Static Type System

In static type system , any method is to be annotated with keyword *atomic*. It uses the theory of right and left movers, first proposed by Lipton to prove the correctness of atomic annotations.

3.1 Theory of LEFT and RIGHT Movers

The type system classifies actions as left or right movers as follows. An execution in which an *acquire* operation A on some lock is immediately followed by an action B of a second thread. Since the lock is already held by the first thread, the action B neither acquires nor releases the lock, and hence the acquire operation can be moved to the right of B without changing the resulting state. Thus the type system classifies each lock *acquire* operation as a right mover. Similarly, consider an action A on one thread that is immediately followed by a *lock release* operation B by a second thread. During A, the second thread holds the lock, and A can neither acquire nor release the lock. Hence the lock release operation can be moved to the left of a without changing the resulting state, and thus the type system classifies lock release operations as left movers.



Finally, consider an access (read or write) to a shared variable declared with the guard annotation guarded by l. This annotation states that the lock denoted by expression l must be held when the variable is accessed. Since type system enforces this access restriction, no two threads may access the field at the same time, and therefore every access to this field is both a right mover and a left mover.

3.2 Verification of Atomicity in Multi-threaded Programs : An Example

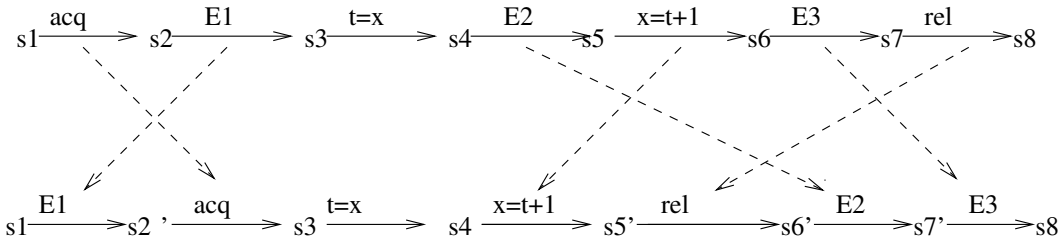
Now consider the following code sequence

- acquires a lock L (the operation acq in the first execution trace in the diagram below);
- reads a variable x protected by that lock L into a local variable t (t=x);
- updates that variable (x=t+1), and;
- releases the lock L (rel).

Suppose that the actions of this method are interleaved with arbitrary actions E1, E2, E3 of other threads. Because the acquire operation is a right mover and the write and release operations are left movers, there exists an equivalent serial execution where the operations of the method are not interleaved with operations of other threads, as illustrated by the following diagram.

Excerpt from java.lang.SrtingBuffer

```
public final class StringBuffer
        public synchronized
       StringBuffer append(StringBuffer sb)
         int len = sb.length();
```




```
        //other threads may change sb.length()
        //so len does not reflect the length of sb
        //sb.getChars(0,len,value,count);
    public synchronized int length()....
    public synchronized void getChars(...).....
```

The *Append* method shown above first calls sb.length(), which acquires the lock sb, retrieves the length of sb, and releases the lock. The length of sb is stored in the variable len. At this point a second thread could remove characters from sb. In this situation, len is now changed and no longer reflects the current length of sb, and so the getChars method is called with an invalid len argument, and may throw an exception. Thus, StringBuffer objects cannot be safely used by multiple threads, even though the implementation is free of race conditions.

More generally, if a method contains a sequence of right movers followed by a single atomic action followed by a sequence of left movers. Then an execution where this method has been fully executed can be reduced to another execution with the same resulting state where the method is executed serially without any interleaved actions by other threads. Therefore, an atomic annotation on such a method is valid.

4 Basic Atomicities

This type system assigns to each expression a type characterizing the value of that expression. In addition, type system also assigns to each expression an atomicity characterizing the behavior or effect of that expression. The set of atomicities includes the following basic atomicities:

- **const :** An expression is assigned the atomicity const if its evaluation does not depend on or change any mutable state. Hence the repeated evaluation of a const expression with a given environment always yields the same result.
- **mover:** An expression is assigned the atomicity mover if it both left and right commutes with operations of other threads. For example, an access to a field F declared as guarded by l is a mover if the access is performed with the lock l held. Clearly, this access cannot happen concurrently with another access to L by a different thread if that thread also accesses F with the lock l held. Therefore, this access both left and right commutes with any concurrent operation by another thread.

5 Runtime Analysis of Atomicity for Multi-threaded Programs

5.1 Atomicity in Transactions

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.

Here are 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, 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.

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
Vector v2 = newVector(v1);
Thread_2
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.

This is free of deadlocks and data races but this does not ensure the absence of all synchronization errors. Let us consider the implementation of Vector in Sun JDK 1.4.2, part of which appears in Figure 1. Let us consider the following execution pattern 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 is less than 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. Type system can ensure that methods are atomic in all possible executions.

Here we study 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. These 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.

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 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. The type 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, 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.

An event E is an instance of one of the operations that are : 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, 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. So algorithm enforces 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 E2 reads the value written by event E1, we call E1 the write-predecessor of E2 in Tr. A read without a write-predecessor in Tr is called an uninitialized read in Tr.

Two traces Tr1 and Tr2 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.

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

5.2 Reduction-Based Algorithm

This atomicity checking algorithm is based on Lipton's reduction theorem.

5.2.1 Commutativity Properties

Following 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 E1 of thread T1 is a lock acquire in a trace, its immediate successive event E2 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 E1 and E2 can be swapped without affecting the result, so E1 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*.

Theorem 1 Lock acquire events are right-movers. Lock release events are left-movers. Race-free reads and race-free writes are both-movers.

Proof. 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.

5.2.2 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. We say that transactions t and t0 have a potential for deadlock if they acquire two locks l1 and l2 in different orders without first acquiring some other lock that prevents their attempts to acquire l1 and l2 from being interleaved. This can be checked with the *good-lock* algorithm. This approach is approximate because it considers only pairs of threads. These observations lead the following theorem.

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*N?L*, where R, L, and N denote right-mover, left-mover, and non-mover, respectively.

The following sections show how to improve above algorithm.

5.2.3 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.

Let us 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 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 2 is not satisfied.

acq(l1)	R(x)	rel(l1)	acq(l2)	R(y)	rel (l2)
acq(l1)	R(y)	rel(l1)	acq(l2)	R(x)	rel(l2)

Theorem 2 can be extended to show that such sets of transactions are atomic. We do this in two steps.

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)*, where AcqRel denotes an acquire of some lock immediately followed by a release of the same lock.

Proof. Based on Theorem 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. 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.

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.

5.2.4 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. Lockset algorithm 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. The 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) := Intersection(Lockset(x),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. 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 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, it 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.

This algorithm 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.

The Multi-lockset algorithm proposes approach , which is more accurate than the preceding algorithms. It incurs higher overhead but is still practical, according to the experimental results. The main three improvements are:

For each variable x, it maintains:

- ReadSets(x), which contains 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 S1 and S2 such that S1 ⊂S2, we remove S2.
- 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) := Intersection of (WriteSet(x), LocksHeld(t)).

ReadSets(x) and WriteSet(x) are not updated by accesses to x before x escapes. Let t1(x) denote the first thread that accesses x after x escapes, or null if there is no such thread. If no thread is concurrent with t1(x) (technique to determine whether two threads are concurrent is described in next section), then accesses to x by t1(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. This algorithm is more accurate than previous lockset-based algorithms.

5.2.5 Other Improvements

The classification of all lock acquires and releases as right-movers and left-movers, respectively, in previous section can be refined. In the following cases, they are as both-movers.
Re-entrant locks: If the thread already holds the lock, an acquire andthe 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 l2 is protected by lock l1 if, whenever a thread holds l2, it also holds l1. Acquire and release by a thread t on a protected lock l2 are both-movers, because adjacent operations of other threads cannot be operations on l2 (because t holds l1).

5.3 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. Atomizer 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 and is labelled with the acquired lock.

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. So we look for unserializable patterns of events. Algorithms are presented for three different cases:(1) multiple transactions that share exactly one variable (2) two transactions that share multiple variables, (3) and multiple transactions that share multiple variables. Locks are not counted as shared variables.

6.1 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.

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

Figure 3: Code Sequence

T is atomic if no feasible interleaving of events of T matches any of these patterns.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 E1 and E2 in transaction t with var(E1) = var(E2), call the variable v, there is a block for E1 and E2 if one of the following conditions holds:

A. If t contains a write to v that precedes E2, then E1 is the last write to v that precedes E2 in t; otherwise, if t contains a read of v that precedes E2, then E1 is the last read of v that precedes E2 in t.
B. If E2 is the final write to v in t, then E1 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 E1 and E2 satisfy one of these conditions, then the block for E1 and E2 is a tuple {op(E1),op(E2),

op(E2), fw(E1), fw(E2), held(E1), held(E2), held(E1, E2)}, 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(E1, E2) is the set of locks held continuously from E1 to E2.
For example, the transaction **t** : acq(l1) R(v) acq(l2) W(v) R(v) rel (l2) rel (l1) (l2) has two blocks,
{R(v),W(v), false, true, l1, l1, l2, l1}
{W(v),R(v), true, false, l1, l2, l1, l2, l1, l2}.

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 E1 and E2 of another block, we check whether held(E) and held(E1, E2) is empty. This simple test is accurate provided there is no potential for deadlock in the program. So we check potential for deadlock as part of the block-based algorithm. To see that this test may be inaccurate if there is potential for deadlock, It is to be noted that E cannot occur between E1 and E2 in the following example, even though inersection of held(E) and held(E1,E2) = **{ }**

t : acq(l1) acq(l2) rel (l2) E rel (l1)
t': acq(l2) acq(l1) rel (l1) E1 E2 rel (l2)

Two blocks b and b' for transactions of different threads are atomic, denoted isAtomicBlk(b, b0), 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 lemma.

Lemma : Let t and t' be transactions that share exactly one variable, with thread(t) not equal to thread(t'). t, t' is atomic iff for every b element of blocks(t) and for every b' which is element of blocks(t'), isAtomicBlk(b, b') holds.

6.2 Two Transactions That Share Multiple Variables

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

A 2-block for a transaction t is a tuple {op(E1), op(E2), held(E1), held(E2), held(E1, E2), heldmid(E1, E2)}formed from two read or write events E1 and E2 of t such that E1 precedes E2 in t, var(E1) is not equal to var(E2), and E1 and E2 are in FW(t) union UR(t). Let 2-blocks(t) denote the set of 2-blocks for transaction t. For example, for the following transaction t, UR(t) = R1(x), FW(t) = W3(x),W4(y), and 2-blocks(t) contains {R1(x),W4(y), null, null,null , null} and {W3(x),W4(y), null, null, null, null}.

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.

0 read	W(x) W(y) W(x)W(y)	W(x) W(y) W(y) W(x)
1read	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)

Figure 4: Transaction sequence

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

Theorem 5 Let t and t' be transactions with thread(t) not equal to thread(t'). t, t' is atomic iff
(i) for every b element of blocks(t), for every b' element of blocks(t'), isAtomicBlk(b, b') holds and
(ii) for every b elemnet of 2-blocks(t), for every b' element of 2-blocks(t'), isAtomic2Blk(b, b') holds.
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 3 is O(E4).

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)

t1: W(x) W(y)
t2: R(x) W(z)
t3: R(z) R(y)

In any potential serial trace equivalent to this one, t1 must precede t2, t2 must precede t3, and t3 must precede t1. Due to the cyclic dependency, no equivalent serial trace exists. Therefore, t1, t2, t3 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. 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 3 Let T be a set of transactions. T is atomic iff for every t, t' element of T, if thread(t) not equal to thread(t'), then
(i)for every b element of blocks(t) and for every b' element of blocks(t'), isAtomicBlk(b, b') holds;
(ii) for every b element of 2-blocks(t) and for every b' element of 2-blocks(t'), isAtomic2Blk(b, b') holds;and
(iii) for every tr element of traces(UR-FW(T)), tr is serializable.

7 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. For a small example of this, consider the threads t1, t2 and t3 in previous example. Only x is shared, so the algorithm in Section 6.1 applies. The blocks are {R(x),W(x), false, true, , o1, o2, {} }, {R(x), dummy, false, false, o2, {}, }, and {R(x), dummy, false, false, {o1}, {}, {}}. The block-based algorithm shows that {t1, t2, t3} is atomic. Where as reduction-based algorithm reports a false alarm for this example.

8 Instrumentation

This section describes the instrumentation of the source code. The instrumentation intercepts the following events:

- reads and writes to all monitored fields.
- entering and exiting synchronized blocks.
- entering and exiting methods that are considered as transactions.
- 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. 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. 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 this 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 a different implementation of the above map is used. For each field F with array type, 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, it 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][0..99] and the cutoff is 3, then only the sub-array [0..2][0..2] is monitored.

9 Conclusion

The Reduction-based algorithm and Block-based algorithm can be used to dynamically check atomicity of programs.This report presents improvements in the accuracy and efficiency of programs.Experimental results also indicate that the majority of methods in benchmarks are atomic, supporting hypothesis that atomicity is a standard methodology in multi-threaded programming.

This report discusses following improvements in tool Atomizer,which implements the on-line reduction-based algorithm.

- Off-line checking,which avoids missing atomicity violations due to miss-classification of events.
- More accurate treatment of accesses to thread-local and read-only variables.
- A new multi-lockset algorithm that produces fewer false alarms than previous lockset algorithms.
- On implementation side , proposed system analyses array,where as Atomizer does not.

Model checking an also be used to check atomicity. Model checking provides stronger guarantees than run-time monitoring algorithms, because it explores all possible behaviors of a program, but model checking is feasible with programs with relatively small state space.

References

[1] www.cs.williams.edu/ freund/papers/04-popl.ps

[2] www.haifa.il.ibm.com/workshops/padta04-3.pdf

[3] www.ce.ucsb.edu/Seminars/Qadeerseminarbw.pdf

[4] www.cse.ucsc.edu/ cormac/papers/popl04.ps