Dynamic Detection of Atomic-Set-Serializability Violations

Christian Hammer†‡, Julian Dolby†‡, Mandana Vaziri†‡, Frank Tip‡
†Universität Karlsruhe (TH) 76128 Karlsruhe, Germany hammer@ipd.info.uni-karlsruhe.de
‡IBM T. J. Watson Research Center P.O. Box 704, Yorktown Heights, NY 10598, USA {dolby,mvaziri,ftip}@us.ibm.com

1. INTRODUCTION

As multi-core systems are coming into general use, concurrency-related bugs are a more significant problem for mainstream programmers. The traditional correctness criterion for concurrent programs is the absence of data races, which occur when two threads access the same shared variable, one of the accesses is a write, and there is no synchronization between them. In general, data-race freedom does not guarantee the absence of concurrency-related bugs. Therefore, different types of errors and correctness criteria have been proposed, such as high-level data races [34], stale-value errors [10], and several definitions of serializability (or atomicity) [16, 17, 35, 14, 34, 2, 15, 30, 22, 37, 36]. According to these definitions of serializability, an execution of read and write events performed by a collection of threads is serializable if it is equivalent to a serial execution, in which each thread’s transactions (or atomic sections) are executed in some serial order. These correctness criteria ignore relationships that may exist between shared memory locations, such as invariants and consistency properties, and therefore may not accurately reflect the intentions of the programmer for correct behavior, resulting in missed errors and false positives.

In previous work [32] we presented a correctness criterion for concurrent systems that takes such relationships into account, and developed an automated inference technique for placing synchronization correctly. The criterion is based on atomic sets of memory locations that must be updated atomically, and units of work, fragments of code that, when executed sequentially, preserve the consistency of the atomic sets that they are declared on. Atomic-set serializability, our correctness criterion, states that units of work must be serializable for each atomic set they operate on. Executions that are not atomic-set serializable can be characterized by a set of problematic data access patterns [32].

In this paper, we present a dynamic analysis for detecting violations of atomic-set serializability in executions of existing Java applications, by checking for the presence of the problematic data access patterns that we previously identified. Our approach provides the following benefits: First, atomic-set serializability is more flexible than existing correctness criteria because it can be used to check for traditional data races [27] (single-location atomic sets), standard notions of serializability (all locations in one atomic set), and a range of options in between. In particular, concurrency bugs such as stale-value errors [10] and inconsistent views [3] can be viewed as violations of atomic-set serializability. A second benefit of atomic-set serializability is that it permits certain non-problematic interleaving scenarios that are

ABSTRACT

Previously we presented atomic sets, memory locations that share some consistency property, and units of work, code fragments that preserve consistency of atomic sets on which they are declared. We also proposed atomic-set serializability as a correctness criterion for concurrent programs, stating that units of work must be serializable for each atomic set. We showed that a set of problematic data access patterns characterize executions that are not atomic-set serializable. Our criterion subsumes data races (single-location atomic sets) and serializability (all locations in one set).

In this paper, we present a dynamic analysis for detecting violations of atomic-set serializability. The analysis can be implemented efficiently, and does not depend on any specific synchronization mechanism. We implemented the analysis and evaluated it on a suite of real programs and benchmarks. We found a number of known errors as well as several problems not previously reported.

Categories and Subject Descriptors
D.1.3 [Programming Techniques]: Concurrent Programming—Parallel Programming; D.2.5 [Software Engineering]: Testing and Debugging—Tracing; D.2.4 [Software Engineering]: Software/Program Verification—Reliability; F.3.2 [Logics and Meaning of Programs]: Semantics of Programming Languages—Program Analysis

General Terms
Algorithms, Experimentation, Measurement, Reliability

Keywords
Concurrent Object-Oriented Programming, Data Races, Atomicity, Serializability, Dynamic Analysis

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 2008 ACM 978-1-60558-079-1/08/05 ...$5.00.
rejected by standard notions of serializability. Third, the problematic data access patterns we check [22] do not depend on specific synchronization constructs such as locks. Our analysis can therefore be used in settings where many existing approaches cannot, such as classes from the Java 5 java.util.concurrent library (e.g., ArrayBlockingQueue and ConcurrentHashMap) and lock-free algorithms. Fourth, since the analysis checks for problematic data access patterns, it only needs to consider fragments of the execution at a time. The entire execution is not needed to detect atomic-set serializability violations.

Key steps of our technique include:

- Using a simple static escape analysis [3] to detect fields of objects that may be accessed by multiple threads,

- For each shared field, maintaining a set of state machines that determine to what extent each problematic interleaving pattern has been matched during execution, and

- Instrumenting the code with yields to encourage problematic interleavings, a technique also known as noise making [6] (this last step is optional).

We implemented the analysis using the Shrike bytecode instrumentation component of the WALA program analysis infrastructure [1]. Our tool instruments the bytecodes of an instrumentation component of the WALA program analysis infrastructure [1]. Our tool instruments the bytecodes of an application in order to: (i) intercept accesses to shared data, (ii) update the state machines accordingly, and (iii) maintain a dynamic call graph to determine the units of work to which these accesses belong. We show how the state machines can be represented efficiently, minimizing the perturbation caused by executing instrumentation code. For our prototype, we made the heuristic assumptions that method boundaries delineate units of work, and that there is one atomic set for (each instance of) each class, containing all the instance fields of that class.

We evaluated the tool on a number of benchmarks, including classes from the Java Collections Framework, and applications from the ConTest suite [12]. We found a significant number of violations, including known problems [12,16], as well as problems not previously reported. Our tool may report false positives for a number of reasons, in particular when the determined units of work are too large, but we found that a significant percentage of the serializability violations (89%) reported by the tool are indeed harmful. On average over all benchmarks, the instrumentation inserted by our tool slows down program execution by a factor of 14, which is similar to, or better than the performance overhead incurred by other dynamic serializability violation detection tools [11,36,37,55,22].

In summary, the contributions of this paper are as follows:

- We present a dynamic analysis for detecting atomic-set serializability violations.

- We implemented the technique using the WALA infrastructure, and demonstrated its effectiveness on a number of Java benchmarks. We found both known bugs and problems not previously reported.

- Our approach is independent of the synchronization constructs employed, and can be applied in situations where many previous techniques cannot (e.g., lock-free algorithms and classes from java.util.concurrent).

2. BACKGROUND

In this section, we present our notion of atomic-set serializability and compare it to several existing notions of serializability and atomicity. Figure 1(a) shows a class Account that declares fields checking and savings, as well as a method transfer() that models the transfer of money from one to the other. Also shown is a class Global that declares a field opCounter that counts the number of transactions that have taken place. For the purposes of this example, we assume that the programmer intends the following behavior:

1. Intermediate states in which the deposit to checking has taken place without the accompanying withdrawal from savings cannot be observed.

2. Concurrent executions of inc() are allowed provided that variable opCounter is updated atomically.

To this end, transfer() and inc() are protected by separate locks, which is accomplished by making each of these methods synchronized. Figure 1(a) also shows a class Test that creates two threads T1 and T2 that execute Account.transfer() and Global.inc() concurrently.

Figure 1(b) depicts an execution in which two threads, T1 and T2, concurrently execute the transfer() and inc() methods, respectively. In particular, we show the various read (R) and write (W) events performed on fields checking (c), savings (s), and opCounter (o). For convenience, each method execution is labeled with a distinct number (1 through 4) in Figure 1(b), and each read/write event is labeled with a sequence of numbers corresponding to the methods on the call stack during its execution. For example, R1(c) indicates the read of field checking during the execution of transfer(), and W1,2(o) denotes the write to field opCounter during the first execution of inc() that was invoked from transfer(). Observe that, in Figure 1(b), the execution of inc() by T2 occurs interleaved between that of the two calls to inc() by T1.

2.1 Atomicity/Serializability

We now discuss several notions of atomicity and serializability that have been defined previously.

Atomicity. Atomicity is a non-interference property in which a method or code block is classified as being atomic if its execution is not affected by and does not interfere with that of other threads. In this setting, the idea is to show that checking and savings are updated atomically by demonstrating that the transfer() method is an atomic section or a transaction.

Lipton’s theory of reduction [21] is defined in terms of right-movers and left-movers. An action b is a right-mover if, for any execution where the action b performed by one thread is immediately followed by an action c performed by a concurrent thread, the actions b and c can be swapped without changing the resulting state [14]. Left-movers and both-movers are defined analogously. In this theory, lock acquires are right-movers, and lock releases are left-movers.

Accesses to shared variables that are consistently protected by some lock are both-movers, and accesses to variables that are not consistently protected by some lock are non-movers. The pattern consisting of a sequence of right movers, followed by at most one non-mover, followed by a sequence of left movers, can be reduced to an equivalent serial execution. However, method transfer() corresponds to the
class Account {
    int checking, savings;
    public Account(int i, int j){
        checking = i; savings = j;
    }
    synchronized void transfer(int n){
        Global.inc();
        savings -= n;
        Global.inc();
    }
    public Account(int i, int j){
        checking = i; savings = j;
    }
    synchronized void transfer(int n){
        Global.inc();
        savings -= n;
        Global.inc();
    }
}

class Global {
    static int opCounter = 0;
    static synchronized void inc(){
        opCounter++;
    }
}

class Test {
    public static void main(String[] args){
        final Account x = new Account(4,7);
        Thread T1 = new Thread(){
            public void run(){
                Global.inc();
            }
        };
        Thread T2 = new Thread(){
            public void run(){
                Global.inc();
            }
        };
        T1.start(); T2.start();
    }
}

sequence: a right-mover (lock acquire at the beginning of
transfer() \(^1\)), 2 both-movers (read/write to field checking),
a right mover (lock acquire at the beginning of inc() \(^3\)), 2
both-movers (read/write to opCounter), a left mover (lock
release at the end of inc() \(^2\)), 2 both-movers (read/write to
savings), a right mover (lock acquire at the beginning of
inc() \(^3\)), 2 both-movers (read/write to opCounter), a left
mover (lock release at the end of inc() \(^3\)), and a left-mover
(lock release at the end of transfer() \(^5\)). Hence, accord-
ing to Lipton’s theory, the method transfer() of Figure 4
is not atomic. In other words, the theory cannot show that
the transfer from checking to savings is performed without
exposing intermediate states to other threads.

Conflict-serializability. Two events that are executed
by different threads are a conflicting pair if they operate on
the same location and one of them is a write. Two executions are conflict-equivalent \([7,36]\) if and only if they contain the same
events, and for each pair of conflicting events, the two
events appear in the same order. An execution is conflict-
serializable if and only if it is conflict-equivalent to a serial
execution. For the threads T1 and T2 in our example, two
serial executions exist, as shown in Figure 4(c) and (d). The execution of Figure 4(b) is not conflict-equivalent to either of these because:

- The pairs of conflicting events include: \((R_{1,2}(o),W_{4}(o)),
\)(R_{1,3}(o),W_{4}(o)), (W_{1,2}(o),R_{2}(o)), and (W_{1,3}(o),R_{1}(o)).

- In order for execution (b) to be conflict-equivalent to
serial execution (c), both \(R_{1,2}(o)\) and \(R_{1,3}(o)\) must
occur before \(W_{4}(o)\), and both \(W_{1,2}(o)\) and \(W_{1,3}(o)\) must
occur after \(R_{4}(o)\). This is not the case either.

Hence, execution (b) is not conflict-serializable.

View-serializability. Two executions are view-
equivalent \([7,36]\) if they contain the same events, if each
read operation reads the result of the same write opera-
tion in both executions, and both executions must have the
same final write for any location. An execution is view-
serializable if it is view-equivalent to a serial execution. It is
easy to see that execution (b) is not view-equivalent to serial
execution (c), because there, \(R_{4}(o)\) reads from \(W_{1,3}(o)\),
whereas in execution (b), it reads from \(W_{1,2}(o)\). Likewise,
execution (b) is not view-equivalent to serial execution (d)
because there, \(R_{1,3}(o)\) reads from \(W_{1,2}(o)\), whereas in
execution (b), it reads from \(W_{4}(o)\). Hence, execution (b) is not
view-serializable. View- and conflict-serializability differ
only on how they treat blind writes, i.e., when a write
performed by one thread is interleaved between writes to
the same location by another thread. Conflict-serializability
implies view-serializability \([7,36]\).

2.2 Atomic-set serializability

The existing notions of atomicity, conflict-serializability
and view-serializability reject the non-problematic execution
of Figure 1(b) because these notions do not take into
account the relationships that exist between memory loca-
tions. Atomic-set serializability assumes the existence of
programmer-specified atomic sets of locations that must be
updated atomically, and units of work on an atomic set, code
fragments that, when executed sequentially, preserve consis-
tency of the atomic set. Given assumption (1) stated above,
we assume that checking and savings form an atomic set
\(S_1\), and that transfer() \(^1\) is a unit of work on \(S_1\). Moreover,
from assumption (2) stated above, we infer that opCounter
is another atomic set \(S_2\) and Global.inc() \(^2\), Global.inc() \(^3\),

Figure 1: (a) Example program. (b)–(d) Three different thread executions.
and \(\text{Global.inc}()\) \(^1\) are units of work on \(S_2\). Atomic-set serializability is equivalent to conflict serializability after projecting the original execution onto each atomic set, i.e., only events from one atomic set are included when determining conflicts.

The projection of execution (b) onto atomic set \(S_1\) contains the following sequence of events:

\[ R_1(c)\ W_1(c)\ R_1(s)\ W_1(s) \]

This is trivially serial, because the events from only one thread are included. Furthermore, the projection of execution (b) onto atomic set \(S_2\) is:

\[ R_{1,2}(a)\ W_{1,2}(a)\ R_4(o)\ W_4(a)\ R_{1,3}(o)\ W_{1,3}(o) \]

which is also serial because the events of units of work \(\text{Global.inc}()\), \(\text{Global.inc}()\), and \(\text{Global.inc}()\) \(^4\) are not interleaved. Therefore, execution (b) is atomic-set serializable.

In conclusion, by taking the relationships between shared memory locations (atomic sets) into account, atomic-set serializability provides a more fine-grained correctness criterion for concurrent systems than the traditional notions of Lipton-style atomicity, conflict-serializability, and view-serializability. In practice, conflict- or view-serializability and atomicity would classify execution (b) as having a bug, but atomic-set serializability correctly reveals that there is none. On the other hand, if a coarser granularity of data is desired or available, all three locations can be placed in a single atomic set, in which case our method would revert to the traditional notion of conflict-serializability.

3. ALGORITHM

Let \(L\) be the set of all memory locations. A subset \(L \subseteq L\) is an atomic set, indicating that there exists a consistency property between those locations. For two locations \(l\) and \(l'\), we write \(\text{sameSet}(l, l')\) to indicate that \(l\) and \(l'\) are in the same atomic set. An event is a read \(R(l)\) or a write \(W(l)\) to a memory location \(l \in L\), for some atomic set \(L\). We assume that each access to a single memory location is uninterrupted. Given an event \(e\), the notation \(\text{loc}(e)\) denotes the location accessed by \(e\).

A unit of work \(u\) is a sequence of events, and is declared on a set of atomic sets. Let \(U\) be the set of all units of work. We write \(\text{sets}(u)\) for the set of atomic sets corresponding to \(u\). We say that \(\bigcup_{u \in \text{sets}(u)} L\) is the dynamic atomic set of \(u\). Units of work may be nested, and we write \(u -\ u'\) to indicate that \(u'\) is nested in \(u\). Units of work form a forest via the \(-\) relation.

An access to a location \(l \in L\) appearing in unit of work \(u\) belongs to the top-most (with respect to the \(\text{-}\) forest) unit of work \(u'\) within \(u\) such that \(L \in \text{sets}(u')\). The notation \(R_u(l)\) denotes a read belonging to \(u\), and similarly for writes. If a method \(\text{foo}\) calls another method \(\text{bar}\), where both are declared units of work for the atomic set \(L_1\) and \(\text{bar}\) reads a location \(l \in L_1\) in \(\text{bar}\), then this read belongs to \(\text{foo}\), as \(\text{foo} \leftarrow \text{bar}\). Given an event \(e\), the notation \(\text{unit}(e)\) denotes the unit of work of \(e\).

A thread is a sequence of units of work. The notation \(\text{thread}(u)\) denotes the thread corresponding to \(u\). An execution is a sequence of events from one or more threads. Given an execution \(E\) and an atomic set \(L\), the projection of \(E\) on \(L\) is an execution that has all events on \(L\) in \(E\) in the same order, and only those events.

<table>
<thead>
<tr>
<th>Data Access Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. (R_u(l)\ W_{u'}(l)\ W_u(l))</td>
<td>Value read is stale by the time an update is made in (u).</td>
</tr>
<tr>
<td>2. (R_u(l)\ W_{u'}(l)\ R_u(l))</td>
<td>Two reads of the same location yield different values in (u).</td>
</tr>
<tr>
<td>3. (W_u(l)\ R_{u'}(l)\ W_u(l))</td>
<td>An intermediate state is observed by (u').</td>
</tr>
<tr>
<td>4. (W_u(l)\ W_{u'}(l)\ R_u(l))</td>
<td>Value read is not the same as the one written last in (u).</td>
</tr>
<tr>
<td>5. (W_u(l)\ W_{u'}(l)\ W_u(l))</td>
<td>Value written by (u') is lost.</td>
</tr>
<tr>
<td>6. (W_u(l)\ W_{u'}(l)\ W_u(l))</td>
<td>Memory is left in an inconsistent state.</td>
</tr>
<tr>
<td>7. (W_u(l)\ W_{u'}(l)\ W_u(l))</td>
<td>same as above.</td>
</tr>
<tr>
<td>8. (W_u(l)\ W_{u'}(l)\ W_u(l))</td>
<td>same as above.</td>
</tr>
<tr>
<td>9. (W_u(l)\ R_{u'}(l)\ R_u(l))</td>
<td>State observed is inconsistent.</td>
</tr>
<tr>
<td>10. (R_u(l)\ W_{u'}(l)\ W_u(l))</td>
<td>same as above.</td>
</tr>
<tr>
<td>11. (R_u(l)\ W_{u'}(l)\ W_u(l))</td>
<td>same as above.</td>
</tr>
<tr>
<td>12. (R_u(l)\ W_{u'}(l)\ W_u(l))</td>
<td>same as above.</td>
</tr>
<tr>
<td>13. (R_u(l)\ W_{u'}(l)\ R_u(l))</td>
<td>same as above.</td>
</tr>
<tr>
<td>14. (W_u(l)\ R_{u'}(l)\ R_u(l))</td>
<td>same as above.</td>
</tr>
</tbody>
</table>

Figure 2: Problematic Data Access Patterns.

A data access pattern is a sequence of events that originate from two or more threads. For example, \(R_u(l)\ W_{u'}(l)\ W_u(l)\) is a data access pattern where unit of work \(u\) first reads \(l\), then another unit of work \(u'\) performs a write, followed by a write by \(u\). An execution is in accordance with a data access pattern if it contains the events in the data access pattern, and these appear in the same order.

Figure 2 shows a number of problematic data access patterns (taken from [32] where 3 patterns were pairwise symmetric) in which data may be read or written inconsistently. Definition 1 below defines a data race in terms of the problematic data access patterns of Figure 2.

**Definition 1 (Data Race).** Let \(L\) be an atomic set, \(l_1, l_2 \in L\), \(l\), one of \(l_1\) or \(l_2\), and \(u\) and \(u'\) two units of work for \(L\), such that \(\text{thread}(u) \neq \text{thread}(u')\). An execution has a data race if it is in accordance with one of the data access patterns of Figure 2.

As an example, consider Figure 2(a) which shows two threads \(T_1\) and \(T_2\) with associated units of work \(u_1\) and \(u_2\), respectively, which operate on two shared locations, \(x\) and \(y\). Figure 2(b) shows an execution in which the following sequence of operations occurs: First (i) \(T_1\) executes its conditional expression, then (ii) \(T_2\) executes its conditional expression, then (iii) \(T_2\) executes the body of its if-statement, and finally (iv) \(T_1\) executes the body of its if-statement. An occurrence of problematic interleaving pattern 11 is highlighted in Figure 2(b) using underlining.

3.1 Race Automata

Our approach for detecting atomic-set serializability violations relies on the construction of a set of race automata...
respectively. The automaton stays in state 0 after the first
pattern corresponding to pattern 11 of Figure 2. To detect
for all other events. Reaching an accept state results in our
bern under consideration, and each state has a self-transition
that are used to match the problematic data access patterns
of Figure 2 during program execution. Each race automaton
has an initial state in which no event of the pattern has been
matched yet, an accept state in which the entire pattern has
been matched, and a number of intermediate states. Transi-
tions between states are labeled with the events of the pat-
tern under consideration, and each state has a self-transition
for all other events. Reaching an accept state results in our
tool issuing a warning. For example, the race automaton
depicted in Figure 3 detects the problematic data access pat-
tern corresponding to pattern 11 of Figure 2. To detect
pattern 11 in the execution of Figure 3(b), u1 and u2 in
the execution are bound to u and u′ in the pattern, and
locations y and x are bound to pattern variables l1 and l2,
respectively. The automaton stays in state 0 after the first
event R_{u1}(x) in the execution of Figure 3(b), which does
not match the first event in the pattern, and transitions to
state 1 when it observes the second event, R_{u1}(y). The next
three events do not change the state, but then W_{u2}(y)
causes a transition to state 2. The next two events, W_{u2}(x)
and R_{u1}(x) cause transitions to states 3 and 4, respectively, at
which point the pattern is fully recognized, and a warning
is issued.

Figure 2 shows that there are 14 patterns that need to be
matched simultaneously. In addition, for each pair of vari-
ables l and l′, there are two ways of matching them against
two program locations, and likewise, there are two ways of
matching the units of work u and u′ against observed units
of work in the execution. Therefore, we need to construct the
14 automata discussed above for each tuple t ∈ \U \times \U \times \L \times \L.

The corresponding automata \( (Q', \Sigma, \delta', S'_0, F') \)
are defined as follows: For each scenario i, \( Q' \) contains states \( S'_j \)
representing that exactly \( j \) events of the scenario have already
been detected, including the accept state \( F' \). The input
alphabet \( \Sigma \) is the union of all traceable events
\[
\Sigma = \bigcup_{u,l \in U} \{ R_u(l), W_u(l) \},
\]
and the transition function \( \delta' : Q' \times \Sigma \rightarrow Q' \) is defined as
follows:
\[
\delta'(S'_j, e) = \begin{cases} S'_{j+1} & \text{if } e \text{ is the } j^\text{th} \text{ event of scenario } i \\ S'_j & \text{otherwise} \end{cases}
\]

Conceptually, when we process an event \( e \), all automata
for all tuples \( q \in \{(u1, u2, l, l') | \text{unit}(e) \in \{(u1, u2) \wedge \text{thread}(u1) \neq \text{thread}(u2) \wedge \text{loc}(e) \in \{(l, l')\}) \} \) need to be up-
dated. While this may require significant space and process-
time in principle, the implementation techniques pre-
seed below make this approach quite feasible in practice.

### 3.2 Efficient Representation of Race Automata

For each tuple \( t \in \U \times \U \times \L \times \L \), there are 14 race automata
that need to be represented. Since each automaton has at
most 5 states, we can represent the dynamic state of an
automaton with 3 bits. For each tuple \( t \), we use a long
value to capture the state of all 14 corresponding automata
(42 bits). Representing automata for all tuples in a program
is prohibitive. So we only capture automata for the tuples
that actually appear during the dynamic analysis and delete
them when they are no longer needed.

We use the notation \((u, u', l, l') \) to represent the set of tu-
ples \( \bigcup_{u,v \in U} \{(u, u', l, l') \} \). Likewise, the notation \((u, u', l, l') \)
denotes the set of tuples \( \bigcup_{u \in U} \{(u, u', l, l') \} \) call such tu-
ples summary tuples. We define a map \( Bits \) which takes a
(summary) tuple and maps it to a bitset containing the
dynamic state of all 14 corresponding race automata. We use
the shorthand notation \( Bits \) to denote the range of the map.

Procedures for manipulating bitsets are the following.
There are two ways of creating a new bitset.

- createBits(u, u', l, l') create a new bitset, initializes it to all
  zeroes, and associates it to tuple \((u, u', l, l')\) in map \( Bits \).
- copyBits(u, u', l, l', b) creates a new bitset, copies the con-
tents of the bitset \( b \) into it, and associates it to tuple
  \((u, u', l, l')\) in map \( Bits \).
- deleteBits(u, u', l, l') deletes the bitset corresponding to tuple
  \((u, u', l, l')\) in map \( Bits \).
- updateBits(b, e) takes a bitset \( b \) and an event \( e \) and updates the states of
  the automata represented by \( b \) according to \( e \).
- reportBits(u, u', l, l') checks if any of the automata associ-
  ated with \((u, u', l, l')\) have reached an accept state and re-
  ports them to the user.

Figure 3 shows pseudocode for our algorithm. The algo-
rithm consists of intercepting events in the execution and
for each event \( e = (u, l, l') \): (i) creating automata if necessary
(\( Create(e) \)); and (ii) updating existing automata (\( Update(e) \)).
At the end of the updates, any automata that have reached
an accept state are reported to the user. GC() runs regularly
to clean up unnecessary bitsets.

Procedure \( Create(e) \) works as follows: If \( e \) is a first occur-
rence of an event in \( u \) on \( l \), then a new bitset is created for
tuple summary \((u, u', l, l')\). The rest of the body of \( Create(e) \)
deals with refining tuple summaries and creating new
bitsets based on their corresponding ones. For example, if \( b =
Bits(u', u', l', l') \) such that \( u \) and \( u' \) are from different threads,
and \( l \) and \( l' \) are different memory locations from the same
atomic set, then event \( e \) causes the creation of a new bitset for
tuple \((u', u', l', l')\). The state of \( b \) is copied into this new
bitset, which is then associated with tuple \((u', u', l', l')\) in map
\( Bits \).

Procedure \( Update(e) \), where \( e \) is an event of \( u \) on \( l \), goes
through the set of bitsets and updates those having \( u \) as

one of their units of work, and \( l \) as one of their memory locations. Finally, procedure \( GC(e) \) works as follows: The set \( T \) represents the set of units of work whose execution has terminated, and is initially empty. Upon termination of each unit of work \( u \), the procedure adds \( u \) to \( T \), and deletes the bitset. It also goes through all bitsets to find those corresponding to tuples with both units of work terminated and deletes them as well, reporting any detected patterns. Note that the bitset corresponding to a summary tuple could reach a final state (detecting one of patterns 1 through 5).

### 4. IMPLEMENTATION

In this section, we present details of our implementation of the algorithm presented in Section 3.1. We first present our choice of defaults for atomic sets and units of work (Section 4.1). Then we discuss how we perform instrumentation to capture events (Section 4.2).

#### 4.1 Defaults for Atomic Sets and Units of Work

We assume that all non-final, non-volatile instance fields of a class (including inherited instance members) are members of a single per-instance atomic set. All accessible non-static public and protected methods in that class, and its superclasses are considered initial units of work declared on these atomic sets. All its non-final, non-volatile static fields form another per-class atomic set with all non-private methods of the class as initial units of work.

Our previous work states a crucial condition: We assume that each access to a member of an atomic set is done within a unit of work declared on that atomic set [32, Section 4.1]. In order to fulfill this requirement, we assume that a method containing a direct access to a field is an additional unit of work for the atomic set the field belongs to.

#### 4.2 Program Instrumentation

We instrument the program to intercept data accesses and to determine what unit of work each access belongs to. To this end, we use the Shrike bytecode instrumentor of the WALA program analysis infrastructure [1].

Before instrumentation, our tool performs a simple static escape analysis [3] to determine possibly shared fields. This analysis determines a conservative set of possibly-escaping fields by computing the set of all types that are transitively reachable from a static field or are passed to a thread constructor. We instrument all non-final and non-volatile fields of such types. In addition, we instrument accesses to arrays, treating each array element as a separate location.

Our tool uses a concurrent, non-blocking queue similar to [19, Section 15.4.2] to store the events of different threads, which guarantees that no user-thread has to wait because of trace collection. Furthermore it timestamps the events in a sequential order which is a prerequisite for detecting the problematic interleaving scenarios. We chose a non-blocking queue to keep the probe effect [15] (i.e., changes to the system’s behavior due to observation) as low as possible, and since, under contention, a blocking queue will show degraded performance due to context-switching overhead and scheduling delays.

Since a field access itself and the recording of that access do not happen atomically, the scheduler could activate another thread between the actual field access and its interception. Nevertheless, the execution obtained is always a valid execution of the program, i.e., it might happen with a possible scheduling. This is because the recording of an event takes place in the same thread as the access itself, and there is no synchronization between them. Hence, any synchronization that applies to the access also applies to the recording. Thus, the intercepted execution must be consistent with the program’s synchronization scheme, so it is a feasible execution.

To determine what unit of work each access belongs to, we keep track of a dynamic call graph, which is essentially the stack trace for each called method. An access to a location in an atomic set belongs to the top-most unit of work declared on that atomic set. To maintain the dynamic call graph, we instrument method entry and exit points. To detect library callbacks, we also instrument invocation points in the program and compare the invocation’s target to the invoked method at the entry. If the target and the called method do not match, a callback has been detected, in which case we start a new unit of work in the called method.

---

1. This case covers both explicit constructor parameters and uses within thread or runnable methods of state defined in an enclosing scope.
As an option, our instrumentation adds yields at certain points in the program to achieve more interleavings, a technique is called noise making. Ben-Asher et al. found that, with a more elaborate noise strategy, the probability of producing a bug increases considerably [6].

5. EVALUATION

We evaluated our analysis on a suite of benchmarks: programs from the ConTest suite [12,11], the W3C’s Java-based Web server Jigsaw (version 2.2.5), and classes from the Java Collections Framework. The ConTest benchmark suite [22,11] consists of short programs of up to 420 lines of code created by students of a class on concurrency. They contain a variety of known non-trivial concurrency-related bugs. Jigsaw is a large program and we analyzed all 939 classes of the main package, omitting other loaded libraries. For the Java Collections Framework, we chose representatives of different synchronization patterns: java.util.Vector, a fully synchronized class containing known bugs; java.util.ArrayList wrapped in a synchronization wrapper1 and classes containing new Java 5 synchronization primitives such as atomic variables and explicit locks like Java 5 synchronization primitives such as atomic variables.

For each collection class analyzed, we generated a test harness that randomly calls its methods 10000 times with appropriate parameters in each of ten threads. Noise making was crucial when executing on a single core processor, but with multicore our exhaustive noise insertion exposed about the same number of races as without (which is consistent with [5]).

Table 1 shows the results of our analysis, where each benchmark was at most executed twice. We evaluated each error report from our tool manually to determine benign (Table 2) and malignant violations. For evaluation purposes, we followed the strict semantics of the new Java memory model (JMM) as described in [20]. We classified all access to shared data (in non-volatile variables) without proper synchronization as a malign bug. We also classified the double-checked-locking anti-pattern as broken when we found it on non-volatile variables. There are, however, executions that exhibit a problematic data access pattern where the code does not exhibit erroneous behavior, largely due to our heuristic choice for units of work. These violations were classified as benign. Note that bugs in the program may manifest themselves as several problematic data access patterns, so there are fewer bugs than serializability violations.

The column “Specified bug found” in Table 1 applies to the documented bugs in the ConTest suite only. It indicates whether the (intended) bug of that program was found by our tool. The column “SF” in Table 1 shows the slowdown factor of our analysis compared to non-instrumented execution. The instrumentation itself imposes a slowdown factor of our analysis compared to non-instrumented executions that exhibit a problematic data access pattern. The instrumentation itself imposes a slowdown factor of 25x [22, block-based]. Regarding our false positive rate of about 11%, our tool improves significantly over previous approaches: With Atomizer [14] 90% of all found serializability violations were no actual bugs, while the best rate was achieved by algorithms in Wang’s work [30] with about 67% false positives. The column “[B]” shows the maximum number of quadruples in the algorithm of Figure 3 before each call to GC(), and the column “Stack depth” shows the average stack depth of intercepted events. No direct correlation between these two numbers and the program size or slowdown factor is evident.

For the ConTest benchmarks, our tool found all the specified bugs for all but 4 programs. For 3 benchmarks, our tool missed the violations due to the heuristics for units of work: For MergeSort, AllocationVector and Shop the actual unit of work was in an unsynchronized method, that called two synchronized methods of another object. Our tool groups only calls to the same target object into one unit of work, unless the calling method accesses fields of the called method’s target directly. To circumvent this problem, future work needs to offer an annotation mechanism for unit-for constructs as presented in our previous work [32]. Atomicity checkers like [30] would miss those three bugs, too, as the transaction boundaries are determined according to synchronized blocks, but the bug in these programs is the fact that synchronization in the caller is missing. Our heuristics miss the bug in FileWriter due to its curious data access pattern: units of work span multiple threads. Our heuristics assume that this is not the case.

In Jigsaw, we found a bug in the extension finishing is updated in the synchronized method shutdown (invoked by some other thread of class org.w3c.jigsaw.daemon. ServerShutdownHook), but read in run without synchronization. To make this code safe, the new JMM requires that this field be volatile, or else the program might never terminate on a multi-core processor. In addition, we found violations that were due to a possible view inconsistency in a synchronized method containing a wait(). We did not find the bug described in [30] in the version we analyzed (2.2.5). When manually inspecting the source code we found that the corresponding code sections are synchronized.

Previous work on atomicity [30] reported a serializability violation in the 1.4 version of Vector: The constructor with a collection argument does not synchronize on the collection parameter. We found this bug still present in Java 5. In java.util.concurrent.AbstractBlockingQueue we discovered a similar bug in the addAll(Collection) method. The JavaDoc of AbstractBlockingQueue, which it inherits, states that the “behavior of this operation is undefined if the specified [parameter] collection is modified while the operation is in progress”. We discovered this bug together with violations for the other “bulk” methods that take parameters of type Collection. Manual inspection found that if the parameter collection is properly synchronized, the other bulk methods’ violations are benign.

We also instrumented ArrayLists in a synchro-
ization wrapper (Collections.synchronizedList(...)). ArrayList’s constructors resemble the code of Vector, so our tool reported the same serializability problem as for the constructor of Vector with a collection parameter. Apart from that, we found all bulk methods (except for addAll(), which is redefined in ArrayList) unserializable when the parameter collection is modified concurrently. The reason for this is that the synchronization wrappers do not provide a synchronized version of the iterator() method but returns an iterator to the backing (unsynchronized) collection.

For LinkedBlockingQueue we found a serializability violation when the inherited addAll(Collection) is executed concurrently with the clear() method. We also found several problematic data access patterns involving the last field. If the queue is cleared while addAll() is being executed the resulting state does not correspond to any serial execution of the two methods. The documentation for this class confirms that behavior is undefined in this case.

In summary, we found our analysis effective for determining atomic-set serializability problems. In Table 1 89% of the reported violations are malign. We ran our analysis on a realistic web server implementation and on typical library code with inner classes and inheritance. All these features are naturally supported by our heuristics.

6. RELATED WORK

We discuss three broad classes of related work: traditional data race detection, high-level data race detection, and detection of violations of atomicity and serializability.

Traditional work on error detection for concurrent programs has been focused on data races. A data race occurs when there are two concurrent accesses to a shared memory location, at least one of which is a write, and there is no synchronization between them. Static approaches for detecting data races include type systems where the programmer indicates proper synchronization via type annotations [8, 13], model checking (see, e.g., [29]), and static analysis [25, 24].

Dynamic analyses for detecting data races include those based on the lockset algorithm [31, 33] on the happens-before relation [24], or on a combination of the two [28]. Savage et al. [31] present a practical implementation of the lockset algorithm in the Eraser tool. The basic idea is lockset refinement: associated with each variable v is the set of locks C(v) that initially consists of all locks. At each access to v, C(v) is intersected with the locks held by the current thread. If C(v) becomes empty, no lock consistently protects v, and a race is reported. The happens-before approach checks whether conflicting accesses to shared data are ordered by explicit synchronization. O’Callahan et al. [23] combined lockset based and happens-before based detection to improve both the overhead and the accuracy of traditional data race detection.

Narayanasamy et al. [25] present a dynamic race detection tool and an automated technique for classifying the races found by the tool as benign or malign. This classification is based on replaying the execution of a piece of code that exhibits a race according to two different executions, and observing whether or not the resulting executions produce different results.

Both static and dynamic approaches to detecting races scale reasonably well for real applications and have detected a large number of bugs in real software [33, 28, 10, 31, 25, 24]. However, a data race is a heuristic indication that a concurrency bug may exist, and does not directly correspond to a notion of program correctness. In our approach, we consider serializability, and in particular atomic-set serializability as a correctness criterion, which captures the programmer’s intentions for correct behavior directly. Moreover, our approach is independent of any synchronization mechanism unlike these techniques.

A program without data races may not be free of concurrency bugs as shown in [10]. These high-level data races may take the form of view inconsistency, where memory is read inconsistently, as well as stale-value errors [9], where value read from a shared variable is used beyond the synchronization scope in which it was acquired. Our problematic data access patterns capture these forms of high-level data races,
as well as several others, in one framework.

In Section 2, we illustrated the differences between atomic-set serializability and atomicity and other forms of serializability. Flanagan and Freund [14] present Atomizer, a dynamic atomicity checker for multi-threaded Java programs, based on Lipton’s theory of reduction [21]. Atomizer uses a variation on Eraser’s LockSet algorithm [31] to determine which shared variables may be involved in data races, and inserts instrumentation code that issues warnings when atomicity violations are detected.

Wang and Stoller present a number of different algorithms for detecting atomicity violations [35, 37, 38]. The Multilockset algorithm [37] improves on the Eraser algorithm [31] by using dynamic escape analysis, happens-before information, and information about held locks. The Reduction-Based Algorithm for checking atomicity [37] resembles Flanagan and Freund’s approach [14], but relies on the Multilockset algorithm for determining variables involved in data races [37]. The Block-Based Algorithm [35, 37] is based on non-serializable interleaving patterns that correspond to our patterns 1–4. Atomicity violations are detected by considering pairs of blocks from different transactions; warnings are issued for matches with one of the unserializable patterns. Wang and Stoller present an extension of this approach to non-serializable interleaving patterns that involve multiple variables. However, they view the heap as a single atomic set, whereas our approach is parameterized by a partitioning of the heap into multiple atomic sets. Wang and Stoller also [35] present two Commit-Node Algorithms for checking view serializability and conflict serializability (detailed comparison presented in Section 2).

Lu et al. [22] detect atomicity violations in C programs. They observe many correct “training” executions of a concurrent application and record nonserializable interleavings of accesses to shared variables. Then, nonserializable interleavings that only arise in incorrect executions are reported as atomicity violations. They only detect atomicity violations that involve a single shared variable, whereas our approach can handle multiple locations. The patterns of nonserializable interleavings in [22] correspond to our patterns 1–4. They view our pattern 5 (two writes interleaved by a write) as serializable, due to the use of a slightly different notion of serializability (view-serializability).

Another serializability violation detector was presented by Xu et al. [35]. It dynamically detects atomic regions (called Computation Units or CUs) using a region hypothesis, which proved useful in their experiments but is not sound in general. Thus, their analysis produces both false positives and negatives. Non-serializability checking is done using a heuristic based on strict two-phase locking. Like our work, it does not rely on the possibly buggy locking structure of the program.

7. CONCLUSIONS AND FUTURE WORK

In previous work [32], we proposed a correctness criterion for concurrent object-oriented programs. This criterion, referred to as atomic-set-serializability in this paper, is more flexible than existing notions of atomicity and serializability because it is parameterized by a programmer-specified partitioning of memory locations into atomic sets. Selecting a partitioning that matches the granularity of a concurrent data structure can help avoid some of the false positives and missed errors associated with existing notions of atomicity and serializability. Moreover, atomic-set-serializability is independent of a specific synchronization mechanism, and can therefore be applied in settings where most other approaches cannot (e.g., lock-free algorithms).

The contributions of this paper are threefold. First, we present a dynamic analysis technique to find violations of atomic-set serializability. Second, we implemented the dynamic analysis in a practical tool that can be applied in realistic scenarios with acceptable overhead. Third, we demonstrated that our tool is capable of detecting a high number of atomic-set serializability problems, including both known errors and problems not previously reported. To the best of our knowledge, we are the first to report concurrency-related problems in classes from the Java 5 concurrent collections framework in package java.util.concurrent.

Currently, our tool uses a fixed set of heuristics for partitioning memory locations into atomic sets. Our present results indicate that, in some cases, our tool fails to find errors when this partitioning is suboptimal. Longer term, we plan to extend our tool to allow users to specify atomic sets using annotations.

Acknowledgments

We thank Stephen Fink for useful discussions about program instrumentation with Shrike, and Jan Vitek, Andreas Lochbihler and Daniel Wasserrab for useful feedback.

8. REFERENCES

detection of race conditions and deadlocks. In Proc.
Towards a framework and a benchmark for testing
tools for multi-threaded programs. Conc. & Comp.: 
[12] Y. Eytani and S. Ur. Compiling a benchmark of
International Parallel & Distributed Processing
Symposium (IPDPS’04), 2004.
detection for java. In PLDI ’00: Proceedings of the
ACM SIGPLAN conference on programming language
design and implementation, 2000.
atomics checker for multithreaded programs. In
POPL ’04: Proceedings of the 31st ACM
SIGPLAN-SIGACT symposium on Principles of
inference for atomicity. In TLDI ’05: Proceedings of
the 2005 ACM SIGPLAN international workshop on
Types in languages design and implementation, pages
[16] C. Flanagan and S. Qadeer. A type and effect system
for atomicity. In PLDI ’03: Proceedings of the ACM
2003 conference on Programming language design and
TLDI ’03: Proceedings of the 2003 ACM international
workshop on Types in languages design and
[19] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes,
and D. Lea. Java Concurrency in Practice. Addison
Language Specification. Addison Wesley Professional,
http://java.sun.com/docs/books/jls/
properties of parallel programs. Commun. ACM,
[22] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO:
Detecting atomicity violations via access interleaving
invariants. In International Conference on
Architectural Support for Programming Languages and
access anomaly detection scheme. In ASPLOS ’91:
Proceedings of the fourth international conference on
architectural support for programming languages and
operating systems, 1991.
for static race detection. In POPL ’07: Conference
record of the 34th ACM SIGPLAN-SIGACT
symposium on Principles of programming languages,
race detection for java. In PLDI ’06: Proceedings of
the 2006 ACM SIGPLAN conference on Programming
language design and implementation, pages 308–319,
and B. Calder. Automatically classifying benign and
harmful data races using replay analysis. In
Proceedings of the ACM SIGPLAN 2007 Conference
on Programming Language Design and Implementation
[27] R. H. B. Netzer and B. P. Miller. What are race
conditions?: Some issues and formalizations. ACM
race detection. In PPoPP ’03: Proceedings of the
ninth ACM SIGPLAN symposium on Principles and
practice of parallel programming, pages 167–178, New
[29] S. Qadeer and D. Wu. Kiss: keep it simple and
sequential. In Proceedings of the ACM SIGPLAN
Conference on Programming Language Design and
Implementation (PLDI’04), 2004.
Automated type-based analysis of data races and
atomicity. In PPoPP ’05: Proceedings of the tenth
ACM SIGPLAN symposium on Principles and
practice of parallel programming, pages 83–94, New
[31] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and
T. Anderson. Eraser: a dynamic data race detector for
synchronization constraints with data in an
object-oriented language. In POPL ’06: Conference
record of the 33rd ACM SIGPLAN-SIGACT
symposium on Principles of programming languages,
In OOPSLA ’01: Proceedings of the 16th ACM
SIGPLAN conference on Object oriented
programming, systems, languages, and applications,
[34] C. von Praun and T. R. Gross. Atomicity violations in
object-oriented programs. Journal of Object
Technology, 3(6):103–122, June 2004. Special issue:
ECOOP 2003 workshop on FTJIP.
atomicity. In Proceedings of the Workshop on Runtime
Verification (RV’03), 2003. Volume 89(2) of Electronic
runtime detection of atomicity errors in concurrent
programs. In Proceedings of the ACM SIGPLAN
Symposium on Principles and Practice of Parallel
Programming, (PPoPP’06), 2006.
[37] L. Wang and S. D. Stoller. Runtime analysis of
[38] M. Xu, R. Bodik, and M. D. Hill. A serializability
violation detector for shared-memory server programs.
In PLDI ’05: Proc. ACM conference on Programming
language design and implementation, 2005.