Accurate and Efficient Runtime Detection of Atomicity Errors in Concurrent Programs *

Liqiang Wang
Computer Science Dept.
State University of New York at Stony Brook
liqiang@cs.sunysb.edu

Scott D. Stoller
Computer Science Dept.
State University of New York at Stony Brook
stoller@cs.sunysb.edu

Abstract

Atomicity is an important 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, executions of procedures (or methods) can be regarded as transactions. Correctness in the presence of concurrency often requires atomicity of these transactions. Tools that automatically detect atomicity violations can uncover subtle errors that are hard to find with traditional debugging and testing techniques.

This paper presents new algorithms for runtime (dynamic) detection of violations of conflict-atomicity and view-atomicity, which are analogous to conflict-serializability and view-serializability in database systems. In these algorithms, the recorded events are formed into a graph with edges representing the synchronization within each transaction and possible interactions between transactions. We give conditions on the graph that imply conflict-atomicity and view-atomicity. Experiments show that these new algorithms are more efficient in most experiments and are more accurate than previous algorithms with comparable asymptotic complexity.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging; D.2.4 [*Software Engineering*]: Software/Program Verification; D.1.3 [*Programming Techniques*]: Concurrent Programming

General Terms Reliability, Algorithms

Keywords concurrent programming, Java, atomicity, data race, serializability

1. Introduction

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

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

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

Flanagan and Qadeer developed a type system for atomicity [8]. It can ensure that methods are atomic in all possible executions. However, type inference for the type system is NP-complete [6], so the type system may require manual annotation of the program.

* This work was supported in part by NSF under Grant CCR-0205376 and CNS-0509230 and ONR under Grants N00014-02-1-0363 and N00014-04-1-0722.

PPoPP'06 March 29–31, 2006, New York, New York, USA.

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

thread_1                          thread_2
 Vector v2 = new Vector(v1);       v1.removeAllElements();
                                   // v1.add(o);
```

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

In [21, 22], we proposed the reduction-based and block-based algorithms for runtime atomicity checking. Runtime analysis is less powerful than static analysis, because it cannot ensure correctness of all unexplored behaviors of the system, but may be more precise (*i.e.*, give fewer false alarms) for the explored behaviors. Furthermore, runtime analysis does not require manual annotations of the code that are often required by type systems; this is a significant practical advantage.

This paper presents novel algorithms, called commit-node algorithms, for runtime checking of atomicity. The algorithms are offline, *i.e.*, when the program terminates, they are applied to recorded information about the execution. The execution is partitioned into units. A *unit* is a sequence of events executed by a single thread. A *transaction* is a unit expected to behave atomically. For example, the sequence of events executed during a method invocation is often considered as a transaction. Our algorithms check whether every trace (*i.e.*, interleaving) of these units is equivalent to a serial trace, where all events in each transaction of these units are consecutive. If so, we say that the transactions are *atomic*; if not, a potential atomicity violation is reported.

The monitor stores the events of each unit (including transactions) in a tree structure, called an *access tree*. Each node in an access tree denotes an access to an escaped variable (*i.e.*, a variable accessible to multiple threads), or a synchronization operation (*e.g.*, lock acquire and release). After the program terminates, the relationships between nodes in different trees are analyzed, and *inter-edges* are added between them to generate a *forest*. A node connected with inter-edges are called *communication node*. A communication node is called a *commit node* if none of its descendants are communication nodes. In a forest, if the access tree for each transaction has only one commit node, then the set of units is atomic. By considering the synchronization, the commit-node algorithms do not merely look for violations of atomicity in the observed execution, but also attempt to determine whether the non-determinism of thread scheduling could allow violations in other executions.

This commit-node algorithms can check two kinds of atomicity, conflict-atomicity and view-atomicity, which are analogous to conflict-serializability and view-serializability in database systems.

Experiments show that these new algorithms are more efficient in most experiments and are more accurate than previous algorithms with comparable asymptotic complexity.

2. Background

This paper focuses on analyzing Java programs, but the techniques can be applied to other languages.

Event. Informally, an *event* is one step in an execution of a program. This paper considers the following operations on events: read and write escaped variables; acquire and release locks; start and join threads; start and exit invocations of methods; and the barrier synchronization operation discussed in Section 7.4. For example, `synchronized`(l) $\{body\}$ in Java indicates two events (in addition to the events performed by the body): acquiring lock l at the entry point and releasing it at the exit point. Two distinct accesses (even using the same operation) to a variable are different events. Let $held(e)$ denote the locks held by the thread executing event e when e is executed.

Transaction Boundaries. Executions of the following code fragments are considered as transactions by default in this paper: non-private methods, synchronized private methods, and synchronized blocks inside non-synchronized private methods; as exceptions, the executions of the `main()` method in which the program starts and the executions of `run()` methods of classes that implement `Runnable` are not considered as transactions, because these executions represent the entire executions of threads and are often not expected to be atomic. Moreover, start, join and barrier operations are treated as unit boundaries, *i.e.*, they separate the preceding events and following events into different units, and are not contained in any unit. We adopt this heuristic because execution fragments containing these operations are typically not atomic and hence are not expected to be transactions. The events not in transactions form non-transactional units. All events in one non-transactional unit have the same thread period id (introduced in Section 7.3). Note that for nested transactions, we check atomicity of only the outermost transactions, since they contain the inner transactions.

Trace. A *trace* tr is a sequence of events. Given $\langle T, E\rangle$, where T is a set of transactions, and E is a set of non-transactional units, *a trace of* $\langle T, E\rangle$ is an interleaving of events from units in $T \cup E$ that is consistent with the original order of events from each thread and with the synchronization events (*e.g.*, no lock is held by multiple threads at the same time). A trace of $\langle T, E\rangle$ must contain all events from units in $T \cup E$ unless the trace ends in deadlock. This paper assumes that E contains no synchronization; this assumption is satisfied if synchronized blocks are considered to be transactions.

Initial Read and Final Write. Let e_x^r and e_x^w denote a read event and a write event to variable x, respectively. e_x^w is the *write-predecessor* of e_x^r in a trace tr if e_x^w is the last write to x that precedes e_x^r in tr. e_x^r is called a *unit-initial read* if e_x^r does not have any write-predecessor in its own unit in all traces. e_x^r is called a *trace-initial read* in trace tr if e_x^r is not preceded by a write to x in tr. Its write-predecessor is defined to be an imaginary write event e_x^{init} at the beginning of the trace. A write event e_x^w is called a *unit-final write* if it is the last write to x in its unit; a write event e_x^w is called a *trace-final write* in a trace if it is the last write to x in the trace.

Conflict-Equivalence. Two traces tr_1 and tr_2 for $\langle T, E\rangle$ are *conflict-equivalent* iff (*i*) they contain the same events, and (*ii*) for each pair of conflicting events, the two events appear in the

same order in both traces. This corresponds to conflict equivalence in transaction processing in database systems [3].

View-Equivalence. Two traces tr_1 and tr_2 for $\langle T, E\rangle$ are *view-equivalent* iff (*i*) they contain the same events, (*ii*) each read event has the same write-predecessor in both traces, and (*iii*) each variable has the same trace-final write event in both traces. This corresponds to view equivalence in transaction processing [3]. It is easy to show that conflict-equivalence implies view-equivalence [3]. But the converse does not hold.

Conflict-Serializability and View-Serializability. A trace of $\langle T, E\rangle$ is *serial* if the events of each transaction of T form a contiguous subsequence of the trace. Note that the events in each non-transactional unit of E are not required to be contiguous. A trace of $\langle T, E\rangle$ is *conflict-serializable* if it is conflict-equivalent to some serial trace of $\langle T, E\rangle$. A trace of $\langle T, E\rangle$ is *view-serializable* if it is view-equivalent to some serial trace of $\langle T, E\rangle$. Conflict-serializability of a trace tr for $\langle T, E\rangle$ can be decided in polynomial time [3]. Let g be the *serialization graph* for tr, which is a directed graph whose nodes are the units of $T \cup E$, and which contains an edge from node t_i to node t_j if $i \neq j$ and some event of t_i precedes a conflicting event of t_j in tr. tr is conflict-serializable iff g does not contain any cycle containing two or more transactions. In contrast, checking view serializability is NP-complete [16].

Conflict-Atomicity and View-Atomicity. $\langle T, E\rangle$ is *conflict-atomic* if every trace of $\langle T, E\rangle$ is conflict-serializable. $\langle T, E\rangle$ is *view-atomic* if every trace of $\langle T, E\rangle$ is view-serializable. It is easy to show that conflict-atomicity implies view-atomicity, but the converse does not hold. As an example, consider $\langle\{t_1, t_2\}, \emptyset\rangle$, where t_1 is $W_1(x)$ $W_2(x)$, and t_2 is $W(x)$. When $t_2.W(x)$ happens between $t_1.W_1(x)$ and $t_1.W_2(x)$, the trace does not have any conflict-equivalent serial trace, hence $\langle\{t_1, t_2\}, \emptyset\rangle$ is not conflict-atomic; but the trace is view-equivalent to a serial trace $t_2.W(x)$ $t_1.W_1(x)$ $t_1.W_2(x)$, and all the other possible traces are serial, hence $\langle\{t_1, t_2\}, \emptyset\rangle$ is view-atomic.

Potential for Deadlock. $\langle T, E\rangle$ has *potential for deadlock* if some trace of $\langle T, E\rangle$ ends in deadlock. A trace that ends in deadlock with some thread in the middle of a transaction is not equivalent to any serial trace. Therefore, this paper assumes that $\langle T, E\rangle$ has no potential for deadlock. This can be checked using the goodlock algorithm [10] or an extension of it [2].

3. Access Forest and Commit-Node Reduction

3.1 Access Tree

During execution of the instrumented program, the monitor records all events for each unit into an *access tree*. In such a tree, each leaf node is called an *access node* and denotes an access to an escaped variable. Each non-leaf node except for the root is called a *synchronization node* and denotes a synchronization block. The root node denotes the whole unit. The local orders of events within a unit are denoted by the order of branches in the tree. An example appears in Figure 2, where $R(v)$ and $W(v)$ (v is x or y) denote a read event and a write event to v, respectively; $acq(l)$ and $rel(l)$ denote an acquire and a release of lock l, respectively. Since each node in an access tree denotes a set of events, a node and the set of events it denotes are used interchangeably in our description.

3.2 Access Forest

An *access forest* consists of a set of access trees and edges called *inter-edges* between access trees from concurrent units. Section 7.4 describes a happen-before analysis to determine whether two units are concurrent. The edges inside each tree are called *tree-edges*. Nodes with an incident inter-edge are called *communication*

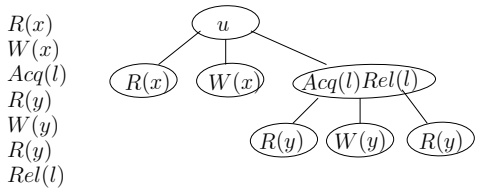

Figure 2. The access tree for a unit u. All events are shown on the left; time increases from top to bottom.

FOR each read event e_x^r
 FOR each write event e_x^w in a concurrent unit
  addInterEdge(e_x^r, e_x^w);

FOR each write event e_x^w
 FOR each write event $e_x^{w'}$ in a concurrent unit
  addInterEdge($e_x^w, e_x^{w'}$);

/ add appropriate inter-edges between nodes of the units containing e and e'. e and e' access the same variable.*/*
PROCEDURE addInterEdge(e, e'){
 IF ($held(e) \cap held(e') = \emptyset$) {
  add an inter-edge between the access node for e
  and the access node for e';
 } ELSE {
  / there must be a common lock in $held(e)$ and $held(e')$, so the next statement finds a suitable node n.*/*
  starting at the root node of the unit that contains e, go
  down the tree along the path to e, until reaching a node
  n corresponding to a synchronization block for a lock l
  in $held(e')$;
  IF ((e is a write) $\vee$ (e is read and not preceded by a write
    to the same variable in the subtree rooted at n)) {
    / otherwise, e is a read and there is a write to the same variable in the subtree rooted at n, so e cannot read the write of e' because of lock l.*/*
   n' = the outermost ancestor of e' corresponding to
        a synchronization block for lock l;
   add an inter-edge between n and n';
}}}

Figure 3. The algorithm to add inter-edges for an arbitrary escaped variable x in the conflict-forest.

nodes; they denote a potential interactions between the corresponding units. Checking conflict-atomicity and view-atomicity require different inter-edges. The access forest used for checking conflict-atomicity is called *conflict-forest*; the access forest used for checking view-atomicity is called *view-forest*.

3.2.1 Conflict-Forest

In the conflict-forest, there are two kinds of relationships denoted by inter-edges between two concurrent units. The first kind of relationship is between a node associated with a write in one of the units and a concurrent node associated with a read to the same variable in the other unit, if the read can read the written value by the write in some trace. The second kind of relationship connects two concurrent nodes associated with two writes to the same variable in the two.

We say "associated with" above because the inter-edge is not necessarily added directly between the access nodes representing those two accesses. Instead, for each pair of accesses satisfying

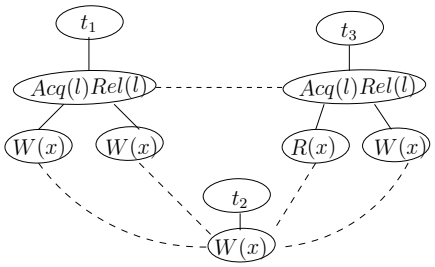

Figure 4. A conflict-forest. The inter-edges are shown as dotted lines.

the above conditions, if there is at least one lock that is held when both operations are performed, then we find the outermost of those common locks, and add an inter-edge between the corresponding synchronization nodes, because this is the granularity at which the parts of the units containing those accesses can be interleaved; if no such lock exists, then an inter-edge is added directly between the access nodes representing those two accesses. By assumption, the set of units does not have potential for deadlock, so the notion of outermost common lock for two accesses is well defined; if there is potential for deadlock, the threads that execute the two accesses could acquire two locks in different orders without first acquiring a common lock.

Intuitively, $\langle T, E\rangle$ is conflict-atomic, if in all traces of $\langle T, E\rangle$, the events of each transaction of T can be repeatedly swapped with adjacent events without affecting the rest of the trace, until the trace is serial, *i.e.*, the events of each transaction are contiguous. If two nodes are connected by an inter-edge, they cannot be swapped. Thus, a node with incident inter-edges is like a non-mover in Lipton's reduction [15, 5, 21, 22].

Figure 3 shows the algorithm to add inter-edges. Figure 4 shows the conflict forest for a set of three units. Note that an inter-edge can denote multiple relationships of the kinds described above. For example, the inter-edge between t_1 and t_3 in Figure 4 denotes two relationships: one is that $t_3.R(x)$ can read the value written by $t_1.W(x)$, and the other is between $t_1.W(x)$ and $t_3.W(x)$.

Besides checking atomicity, the conflict-forest can also be used for detecting data races, since each access node with incident inter-edges indicates a data race.

3.2.2 View-Forest

The view-forest has three kinds of relationships between two concurrent units u_1 and u_2 denoted by inter-edges. (1) The first kind of relationship is between a node of u_1 associated with a write and a node of u_2 associated with a read, if the read can read the written value by the write in some trace. (2) The second kind of relationship connects two nodes associated with two writes to the same variable, respectively, if both writes can be the write-predecessor of the same read in some traces. (3) The third kind of relationship connects two nodes associated with unit-final writes to the same variable.

The algorithm of adding inter-edges for view-forest is shown in Figure 5. It is similar to the algorithm in Figure 3. When adding an inter-edge between a read and its potential write-predecessor, we also add inter-edges between its all potential write-predecessors. $S(e)$ caches all potential write-predecessors for the read e so far. Besides connecting this kind of writes, we also add inter-edges between the unit-final writes to the same variables, instead of adding inter-edges between every two writes to the same variables in conflict forest.

```
FOR each read event e_x^r
 S(e_x^r) = ∅;
 FOR each write event e_x^w in a concurrent unit
  addInterEdge(e_x^r, e_x^w);

FOR each unit-final write event e_x^w
 FOR each unit-final write event e_x^w' in a concurrent unit
  addInterEdge(e_x^w, e_x^w');

/* Note that e and e' access the same variable. */
PROCEDURE addInterEdge(e, e'){
 IF (held(e) ∩ held(e') = ∅){
  add an inter-edge between the access node for e and
  the access node for e';
 } ELSE {
  starting at the root node of the unit that contains e, go
  down the tree along the path to e, until reaching a node
  n corresponding to a synchronization block for a lock l
  in held(e');
  IF ((e is a write) ∨ (e is read and not preceded by a write
     to the same variable in the subtree rooted at n)) {
   n' = the outermost ancestor of e' corresponding to
        a synchronization block for lock l;
   add an inter-edge between n and n';
  } ELSE return;
 }
 IF (e is read) {
  e_w = the preceding write to the same variable and
        in the same unit as e, if any, otherwise null;
  IF (e_w ≠ null)
   S(e) = S(e) ∪ {e_w};
  FOR each e'' in S(e)
   addInterEdge(e',e'');
  S(e) = S(e) ∪ {e'};
}}
```

Figure 5. The algorithm to add inter-edges for an arbitrary escaped variable x in the view-forest.

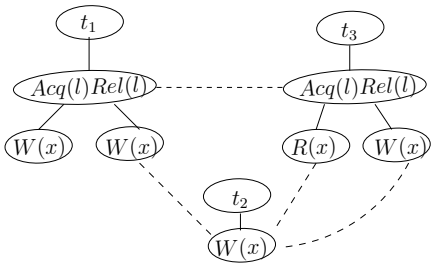

Figure 6. A view-forest. The inter-edges are shown as dotted lines.

Figure 6 shows the view forest after applying the algorithm to the same three units in Figure 4.

3.3 Commit-Node Reduction

Let $n \succcurlyeq n'$ denote that a communication node n contains (*i.e.*, is an ancestor of) another communication node n'. A communication node is called a *commit node* if it is not contained in any other communication nodes. For example, in Figure 6, the communication node "$Acq(l)Rel(l)$" in t_1 contains the communication node "$W(x)$" which is a commit node since it does not contain any other communication nodes.

Intuitively, for a set of nodes $n_1 \geqslant ... \geqslant n_k \geqslant n_c$, the commit node n_c denotes a non-mover, $n_i (1 \leqslant i \leqslant k)$ denotes a larger non-mover which contains n_c. All events of n_1 (which also contains all events of n_2,...,n_c) can be moved to the commit node position through swapping without affecting the other units in all traces. Thus, a transaction with at most one commit node is atomic, but a transaction with two or more commit nodes might be non-atomic. This is described formally in Section 4.

4. The Commit-Node Algorithms for Checking Atomicity

This section presents algorithms for checking conflict-atomicity and view-atomicity.

4.1 Conflict-Atomicity

Theorem 4.1. *Suppose $\langle T, E\rangle$ has no potential for deadlock, and E does not contain any synchronization operations. If each transaction of T has at most one commit node in the conflict-forest, then $\langle T, E\rangle$ is conflict-atomic.*

Proof. To prove that $\langle T, E\rangle$ is conflict-atomic, we need to show that there is a conflict-equivalent serial trace tr' for an arbitrary trace tr of $\langle T, E\rangle$. The general idea is to find a location of some event inside the commit node for each transaction, such that when all events of each transaction are moved to that location, the resulting trace is conflict-equivalent to the original trace. That location is called a *commit point*.

For a commit node n, if n is an access node, its commit point is the location of the access event at tr; if n is a synchronization node, its commit point is any arbitrary location inside the commit node at tr. According to the assumption in the theorem, each communication node contains only one commit node. tr' is constructed from tr as follows: all events of the communication nodes that contain the commit node are moved to the commit point; all events of each transaction not in any communication node are also moved to the commit point of the transaction; all other events are not moved. tr' is serial because every transaction has only one commit node. In the following, we prove that tr' is a legal trace and is conflict-equivalent to tr.

First, we observe that tr' is consistent with the synchronization events. This holds because tr' is serial, and E does not contain any synchronization. So tr' is a trace for $\langle T, E\rangle$.

Next, we show that tr' is conflict-equivalent to tr. Consider conflicting events e_1 and e_2, where e_1 and e_2 occur in units u_1 and u_2 in $T \cup E$, respectively. Without loss of generality, suppose e_1 precedes e_2 in tr. Because e_1 and e_2 conflict, u_1 must contain a communication node n_1 containing e_1, and u_2 must contain a communication node n_2 containing e_2, and n_1 precedes n_2 in tr. After moving e_1 and e_2 to the commit points of u_1 and u_2, respectively, e_1 and e_2 appear at the same order in tr and tr'. Therefore, tr is conflict-equivalent to tr'. □

The condition in Theorem 4.1 for conflict-atomicity is sufficient but not necessary. In Figure 7, the set of transactions is conflict-atomic, even though t_1 contains multiple commit nodes. The following theorem shows that the condition in Theorem 4.1 is an exact test for conflict-atomicity of two transactions.

Theorem 4.2. *Suppose $\langle T, \emptyset\rangle$ has no potential for deadlock, and T contains only two transactions. $\langle T, \emptyset\rangle$ is conflict-atomic iff each transaction in T has at most one commit node in the conflict-forest.*

Proof. A proof sketch appears here; more details are in [23].

"⇐": This direction follows from Theorem 4.1.

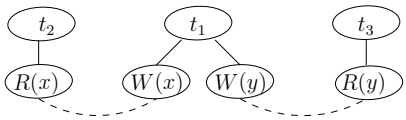

Figure 7. $\langle\{t_1, t_2, t_3\}, \emptyset\rangle$ is both conflict-atomic and view-atomic, but t_1 contains two commit nodes.

"⇒": Suppose $T = \{t, t'\}$. We show that $\langle T, \emptyset\rangle$ is not conflict-atomic if at least one transaction in T has two or more commit nodes. Without loss of generality, suppose t has at least two commit nodes. Let n_1 and n_2 denote two commit nodes of t. t' has at least one commit node. There must be a pair of conflicting events, denoted e_1 and e_1', with $e_1 \in n_1$ and $e_1' \in n_1'$, where n_1' is a communication node in t', and there is an inter-edge between n_1 and n_1'. Similarly, there must be another pair of conflicting events, denoted e_2 and e_2', with $e_2 \in n_2$ and $e_2' \in n_2'$, where n_2' is a communication node in t', and there is an inter-edge between n_2 and n_2'.

If one of n_1' and n_2' contains the other or if $n_1' = n_2'$, we can show that there is a trace tr where n_1' and n_2' happen between n_1 and n_2. Otherwise, we can show that there is a trace tr where n_1' and n_2' happen between n_1 and n_2, or n_1 and n_2 happen between n_1' and n_2'. Hence, tr does not have any conflict-equivalent serial trace. □

For example, in Figure 4, $\langle\{t_1, t_2\}, \emptyset\rangle$ is not conflict-atomic according to Theorem 4.2 because t_1 contains two commit nodes when ignoring t_3. Similarly, $\langle\{t_2, t_3\}, \emptyset\rangle$ is not conflict-atomic, either.

The following theorem gives a more sophisticated (compared to Theorem 4.1) condition to decide conflict-atomicity (for any number of transactions). This theorem (unlike Theorem 4.1) is accurate enough to show that the set of transactions in Figure 7 is conflict-atomic. Note that, when considering cycles in the conflict-forest, tree edges are treated as undirected edges.

Theorem 4.3. *Suppose $\langle T, E\rangle$ has no potential for deadlock, and E does not contain any synchronization operations. If all pairs (if they exist) of communication nodes from the same transaction that do not contain each other are not involved in any cycle of the conflict-forest, then $\langle T, E\rangle$ is conflict-atomic.*

Proof. We prove the contrapositive. Suppose $\langle T, E\rangle$ is not conflict-atomic. Thus, there is a trace tr for $\langle T, \emptyset\rangle$ that does not have any conflict-equivalent serial trace, and there is a directed cycle c in the serialization graph for tr. Suppose that c consists of $\langle t_1, t_2, ..., t_n\rangle$ in order, where $t_1, t_2, ..., t_n \in T$. c implies that there must be an event e_1 of t_1 that happens before an event e_2 of t_2, an event e_2' of t_2 that happens before an event e_3 of t_3, ..., and an event e_n' of t_n that happens before e_1' of t_1 in tr, where e_1 and e_2 access the same variable, e_2' and e_3 access the same variable, ..., and e_n' and e_1' access the same variable. This implies that there is a cycle c' in the conflict-forest. If e_i and e_i' of t_i for $i = 1..n$ are in the same communication node, then both e_i and e_i' happen before e_{i+1} and e_{i+1}' in tr, for $i = 1..n-1$. This contradicts the assumption that e_n' happens before e_1'. Hence, there must be a transaction in $\{t_1, t_2, ..., t_n\}$ that has at least two communication nodes on c' that do not contain each other. □

The *commit-node algorithm* for checking conflict-atomicity works as follows. (1) Instrument the source code of program to be tested as discussed in Section 7.1. (2) Execute the instrumented program, and dynamically construct the conflict-trees. (3) Add inter-edges after the execution terminates. (4) Check the conditions

of Theorem 4.1; if they are satisfied, report that conflict-atomicity holds; otherwise, check the conditions of Theorem 4.3, then report conflict-atomicity holds or not according to whether the conditions are satisfied.

Although this algorithm may report false alarms since the conditions in Theorem 4.1 and Theorem 4.3 are sufficient but not necessary. But we believe that this happens very rarely. In the experiments of Section 8, all the warnings for non-conflict-atomicity reported by the algorithm are confirmed to be true by Theorem 4.2.

Let $|T|$, n_t, and n_e denote the number of transactions, the maximum number of events in a transaction, and the number of events in the whole execution (including non-transactional units), respectively. Theorem 4.3 requires checking, for each pair of communication nodes of the same transaction, whether they are involved in a cycle, i.e., whether each of them is reachable from the other. There are $O(|T| \times n_t^2)$ such pairs, and checking whether two nodes are reachable from each other takes time $O(n_e^2)$, so the worst-case time complexity of the algorithm is $O(|T| \times n_t^2 \times n_e^2)$.

4.2 View-Atomicity

Theorem 4.4. *Suppose $\langle T, E\rangle$ has no potential for deadlock, and E does not contain any synchronization operations. If each transaction of T has at most one commit node in the view-forest, then $\langle T, E\rangle$ is view-atomic.*

Proof. A proof sketch appears here; more details are in [23]. For any trace tr of $\langle T, E\rangle$, we prove that it has a view-equivalent serial trace tr', which is constructed in the same way as in the proof of Theorem 4.1. The definition for the commit point of each transaction is the same as there. By the same reasoning as in that proof, tr' is serial and consistent with the synchronization operations, so tr' is a trace for $\langle T, E\rangle$. Next we prove in two steps that tr' is view-equivalent to tr. (1) We first prove that each read has the same write-predecessor in tr and tr'. The main observation is: if two communication nodes are connected by an inter-edge, then the sets of nodes contained in them cannot be interleaved with each other in any trace. Because there are inter-edges between communication nodes associated with potential write-predecessors for the same read e^r, if one communication node n^w contains the actual write-predecessor of the read e^r in tr, all the communication nodes associated with other potential write-predecessors for e^r must happen in tr before the events represented by n^w or after the commit node associated with e^r. Hence, after moving all events in each transaction to its commit node, e^r has the same write-predecessor in tr' and tr. (2) A similar proof shows that tr and tr' have the same trace-final writes. □

For example, in Figure 6, $\langle\{t_1, t_2\}, \emptyset\rangle$ is view-atomic because each of t_1 and t_2 contains only one commit node in the view-forest. Note that $\langle\{t_1, t_2\}, \emptyset\rangle$ is not conflict-atomic since t_1 has two commit nodes in the conflict-forest.

The condition in Theorem 4.4 for view-atomicity is sufficient but not necessary. In the example of Figure 7, $\langle\{t_1, t_2, t_3\}, \emptyset\rangle$ is view-atomic but t_1 contains multiple commit nodes. The following theorem shows that the condition in Theorem 4.4 is an exact test for view-atomicity of two transactions.

Theorem 4.5. *Suppose T has no potential for deadlock, and T contains only two transactions. $\langle T, \emptyset\rangle$ is view-atomic iff each transaction in T has at most one commit node in the view-forest.*

Proof. "⇐": This implication is justified directly based on Theorem 4.4.

"⇒": We prove the contrapositive, *i.e.*, if at least one of the transactions has at least two commit nodes, then $\langle T, \emptyset\rangle$ is not view-atomic. According to the definition of view-forest, there are three kinds of inter-edge. (1) The first kind of inter-edge denotes the relationship between a read e^r in a transaction t and its potential write-predecessor e^w in the other transaction. If the read has a preceding write e^w_{pre} in its own transaction, then a violation of view-atomicity is possible (because e^w can occur between e^w_{pre} and e^r by the definition of potential write-predecessor), so the desired implication holds. If the read does not have any preceding write in its own transaction, the read and its potential write-predecessor in the other transaction act like a pair of conflicting events, in the sense that their order in a trace determines that the two transactions must follow the order in all serial traces view-equivalent to the trace (this is true with two transactions, although it is not true with more transactions). This is the same property of inter-edges in the conflict-forest that is used in the proof of Theorem 4.2. (2) The second kind of inter-edge denotes the relationship between two writes that are potential write-predecessor for the same read; this indicates a violation of view-atomicity (because either there is a write to some variable x in t' that can occur between a write to x and a read to x in t, or there is a read to x in t' that can occur between two writes to x in t), so the desired implication holds. (3) The third kind of inter-edge denotes the relationship between unit-final writes of each transaction. With two transactions, these final writes also act like conflicting events, in the sense described above, so these edges have the same property as inter-edges in the conflict-forest.

Thus, depending on the kind of edges present, either we immediately conclude that the desired implication holds, or all of the edges have the property of inter-edges in the conflict-forest used in the proof of (⇒) in Theorem 4.2, and the rest of this proof is similar to that proof. □

For example, in Figure 6, $\langle\{t_2, t_3\}, \emptyset\rangle$ is not view-atomic because t_3 contains two commit nodes in the view-forest for t_2 and t_3. The following theorem gives a more sophisticated condition to check view-atomicity.

Theorem 4.6. *Suppose $\langle T, E\rangle$ has no potential for deadlock, and E does not contain any synchronization operations. If all pairs of communication nodes from the same transaction that do not contain each other are not involved in any cycle of the view-forest, then $\langle T, E\rangle$ is view-atomic.*

Proof. The proof is similar to the proof of Theorem 4.3. □

The commit-node algorithm for checking view-atomicity is similar to the algorithm for checking conflict-atomicity proposed in Section 4.1, except that the view-forest is constructed and checked based on Theorems 4.4 and 4.6. Similarly as before, the worst-case time complexity of the algorithm is $O(|T| \times n_t^2 \times n_e^2)$.

5. The Polynomial Equivalence of Conflict- and View-Atomicity

The following theorem shows that the problems of checking conflict-atomicity and checking view-atomicity are polynomially reducible to each other. This result is somewhat surprising, considering that checking conflict serializability is in P [3] and checking view serializability is NP-complete [16], so they are not polynomially reducible unless P=NP. To simply the problem, we consider only transactions *i.e.*, assume $E = \emptyset$; we expect that the result also holds without this restriction.

Theorem 5.1. *The problems of checking conflict-atomicity and checking view-atomicity are polynomially reducible to each other when restricted to problem instances where the set E of non-transactional units is empty.*

Proof. A proof sketch appears here; more details are in [23].

1. We first prove that the problem of checking conflict-atomicity is polynomially reducible to the problem of checking view-atomicity. We transform T as follows. Let l be a lock not used in T. For each variable x, each read e_x^r is replaced by $e_l^{acq}\ e_x^r\ e_l^{rel}$; and each write e_x^w is replaced by $e_l^{acq}\ e_x^r\ e_x^w\ e_l^{rel}$, where e_l^{acq} and e_l^{rel} represent an acquire and a release of l, respectively. Let T' denote the resulting set of transactions.

We prove that T is conflict-atomic iff T' is view-atomic. "$\Rightarrow$": Every trace tr' of T' corresponds to some trace tr of T. Every trace tr of T has a conflict-equivalent serial trace tr_s. We transform tr and tr_s in the same manner that is used to transform T, yielding traces tr' and tr'_s of T', respectively. We show in [23] that tr' is view-equivalent to the serial trace tr'_s. "$\Leftarrow$": For each trace tr' of T' and its view-equivalent serial trace tr'_s, we remove the operations inserted when constructing T' from T. This yields traces tr and tr_s of T. We show in [23] that tr is conflict-equivalent to tr_s.

2. We can check view-atomicity of pairs of transactions in T in polynomial time based on Theorem 4.5. In the following we prove that checking view-atomicity can be reduced to checking conflict-atomicity when all pairs of transactions in T are view-atomic. Because all pairs of transactions in T are view-atomic, all non-unit-final writes and non-unit-initial reads do not interact with other transactions, so we remove them. Let T_f denote the resulting set of transactions. We prove that T is view-atomic iff T_f is conflict-atomic. "$\Rightarrow$": We prove the contrapositive by constructing a non-view-serializable trace of T from a non-conflict-serializable trace of T_f. "$\Leftarrow$": For each trace tr of T, there is a corresponding trace tr_f of T_f which has a conflict-equivalent serial trace tr_f^s. A serial trace tr^s of T that is view-equivalent to tr can be constructed from tr_f^s by restoring the removed write and reads. □

6. Comparison with Other Atomicity Checking Algorithms

6.1 Reduction-Based Algorithms

Reduction-based algorithms for checking conflict-atomicity [8, 5, 22] classify events based on commutativity and apply Lipton's reduction theorem [15]. We briefly describe the reduction-based algorithm in [22], which is more accurate than the others.

An event is a *right-mover* (R) if, whenever it appears immediately before an event of a different thread, the two events can be swapped (*i.e.*, they can be executed in the opposite order without blocking) without changing the resulting state. A *left-mover* (L) is defined similarly. Events not known to be left or right movers are *non-movers* (N). Race-free events are both right and left movers; events with race are non-movers. Lock acquire events are right-movers. Lock release events are left-movers. The reduction-based algorithm is based on the following variant of Lipton's reduction theorem: a set T of transactions is conflict-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, followed by release of the same lock.

The following theorem and example together show that Theorem 4.1 is more accurate than the above reduction theorem for conflict-atomicity.

Theorem 6.1. *If a transaction t has the form $(R + AcqA^*Rel)^*N^?(L + AcqA^*Rel)^*$, then t has at most one commit node in the conflict-forest.*

Proof. According to the algorithm in Figure 3, the block denoted by $AcqA^*Rel$ does not contain any communication node. All synchronization blocks denoted by $R^*N^?L^*$ must be nested. Thus, for any two communication nodes in t that are also synchronization nodes, one is a descendant of the other. t contains at most one non-mover, and it occurs in the inner-most synchronization block. Thus, there is at most one access node in t that is also a communication node, and that communication node is a descendant of all other communication nodes in t. Thus, for any two communication nodes of transaction t, one is a descendant of the other. Hence, t has at most one commit node. □

Now we give an example that is conflict-atomic according to Theorem 4.1 but is wrongly reported to be non-atomic by the reduction theorem. Let $T = \{t_1, t_2\}$, where t_1 consists of $e_x^{r_1}$ followed by $e_x^{w_1}$, and t_2 consists of only $e_x^{r_2}$. t_1 has the form NN which does not match $(R + AcqA^*Rel)^*N^?(L + AcqA^*Rel)^*$, but t_1 and t_2 each have only one commit node.

The commit-node algorithm of Section 4.1 contains the benefits of all the improvements to the reduction-based algorithm described in [22], which include the improvements proposed in [5]. For example, for re-entrant locks, thread-local locks, and protected locks, there is no inter-edge connected to the corresponding synchronization nodes.

Therefore, the reduction-based algorithm is less accurate than the commit-node algorithm of Section 4.1.

6.2 The Block-Based Algorithm

The block-based algorithm [22] checks view-atomicity by considering pairs of *blocks* from different transactions. Intuitively, a block captures the information about two accesses and the associated synchronization that is relevant to atomicity checking. The block-based algorithm constructs blocks from an observed trace and then compares each block with all blocks in other concurrent transactions. If two blocks are found to match certain unserializable patterns, the transactions containing them are not atomic. The unserializable patterns are defined based on view serializability; for example, one unserializable pattern is when a write of one transaction can happen between two continuous reads of another concurrent transaction.

The commit-node algorithm and the block-based algorithm are both exact tests for view-atomicity for two transactions, but the former runs much faster in most programs in the experiments in Section 8. For three or more transactions, the commit-node algorithm is an efficient conservative test that is very accurate in practice based on the experiments in Section 8; the block-based algorithm can provide an exact test but is significantly more expensive.

7. Implementation

We implemented the commit-node algorithms for checking conflict-atomicity and view-atomicity in Java. The implementation consists of three parts: instrumentation, monitoring and off-line analysis. Instrumentation is discussed in Section 7.1. The monitor intercepts all events described in Section 2 and constructs access trees. Each access tree is optimized to discard the redundant accesses, as discussed in Section 7.2. If there are more than two identical access trees, we save only two copies, since the rest are redundant for checking atomicity. A dynamic escape analysis introduced in Section 7.3 is used to determine when a variable escapes. A happen-before analysis introduced in Section 7.4 is used to determine whether two units are concurrent. When the program terminates, the algorithm adds inter-edges between access trees, and then checks conflict-atomicity and view-atomicity using the algorithms in Sections 4.1 and 4.2, respectively.

7.1 Instrumentation

We modify the pretty-printer in the Kopi [14] compiler to insert instrumentation as it pretty-prints the source code. The instrumentation intercepts the following events: (1) reads and writes to all monitored fields (see below); (2) entering and exiting synchronized blocks, including synchronized methods; (3) entering and exiting methods that are considered as transactions (discussed in Section 2); (4) calls to thread `start` and `join`; (5) barrier synchronization.

All non-final fields (with primitive type or reference type) of the specified classes (by default, all classes) are monitored. Accesses to these fields in all methods of all classes are instrumented. Local variables are not monitored, because they are accessed by at most one thread. Our system inserts fields into monitored classes to keep track of *shadow information*, *e.g.*, whether the object has escaped. There is no way to insert fields into array classes in Java, so we maintain a hash table that maps each array reference a to an array with shadow information for each element of a. Monitoring every array element causes large slowdown in some programs, so our system supports "sampling" of arrays, in which only index positions below a user-specified cutoff are monitored.

7.2 Optimization: Trimming the Access Tree

It is not necessary to save all accesses to escaped variables. For access nodes with the same parent node, we preserve only the first two read accesses and the first two write accesses (if they exist) to each escaped variable, because the first two reads and writes to x can represent all discarded accesses for checking (conflict or view) atomicity. The resulting trees and forests are said to be *trimmed*.

Theorem 7.1. *For every $\langle T, E\rangle$, and every hypothesis H about the conflict-forest and view-forest in the theorems of Section 4, H holds for the trimmed forest iff it holds for the untrimmed forest.*

Proof. 1. Consider reads. Let R be a set of three or more reads to the same variable that share the same parent node. It is easy to see that in both conflict forest and view forest, either all of them are connected to a given write in another unit by inter-edges, or none of them are connected to it. Suppose we evaluate H considering only the first two reads in R. It is easy to show that considering additional reads in R either does not generate any additional inter-edges, or the generated edges do not affect H.

2. Consider writes in the conflict-forest. Let W be a set of three or more writes to the same variable that share the same parent node. In the conflict forest, either all of them are connected to a given read or write of another unit by inter-edges, or none of them are connected to it. Similarly as for reads, we can show that considering the third and subsequent writes in W either does not generate any additional inter-edges, or the generated edges do not affect H. For the view forest, if W does not contain a unit-final write, then the reasoning is similar to the previous cases; if W contains a unit-final write, it gets removed and the second write in W becomes the unit-final to that variable; it is easy to verify (for each hypothesis H) that this does not affect H. □

7.3 Dynamic Escape Analysis

Before an object escapes from the thread that created it, all operations on it can be ignored when checking atomicity. An object o escapes in the following scenarios: (1) o is stored in a static field or a field of an escaped object; (2) o is an instance of a thread and the thread is started; (3) o is referenced by a field of another object o', and o' escapes (this leads to cascading escape); (4) o is passed as an argument to a native method that may cause it to escape.

To indicate whether an object has escaped, a boolean instance field `escaped` is added to every instrumented class. Its initial value is `false`. To detect when objects escape, we instrument all method calls, and all stores to static fields, instance fields, and arrays. When an object escapes, it is marked as escaped by setting its `escaped` field to `true`, and all objects to which it refers are marked as escaped (and so on, recursively); Java's reflection mechanism is used to dynamically find those objects. More details appear in [22].

7.4 Happen-Before Analysis

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

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

We use a directed graph to represent the happen-before relations between thread periods. There is a path from the node representing p_1 to the node representing p_2 iff p_1 happens before p_2. More details are in [22].

8. Experiments

We perform experiments with 12 programs. They are `elevator`, `tsp`, `sor`, and `hedc` from [19]; `moldyn`, `montecarlo`, and `raytracer` from the Java Grande Benchmark Suite [12]; `StringBuffer`, `Vector`, `Hashtable`, and `Stack` from Sun JDK 1.4.2; and `jigsaw` from W3C [13]. `elevator` simulates the actions of multiple elevators. `tsp` solves the travelling salesman problem; we run it on the accompanying data file `map14`. `sor` is a scientific computing program which uses barriers rather than locks for synchronization. `hedc` is a Web crawler that searches astrophysics data on the Web. `moldyn`, `montecarlo`, and `raytracer` are computation-intensive parallel programs that compute molecular dynamics, Monte Carlo simulation, and ray tracing, respectively. `jigsaw` is a Web server implemented in Java; we instrument only its packages that are related to HTTP service. Table 1 shows the number of lines of code in instrumented classes, *i.e.*, it excludes code in uninstrumented libraries. For all programs that accept the number of threads as an argument, we use three threads. All experiments are done on a Sun Blade 1500 with a 1GHz UltraSPARC III CPU, 2GB RAM, SunOS 5.8, and JDK 1.4.2.

Table 1 compares the running time, result and storage of the commit-node algorithm for view-atomicity described at the end of Section 4.2 with the off-line reduction-based algorithm [22] and the pairwise block-based algorithm [22]. This version of the block-based algorithm checks view-atomicity of all pairs of transactions; the full block-based algorithm, which checks whether the entire set of transactions is atomic, would be significantly slower. "LOC" is the lines of code. "Base time" is the running time of the uninstrumented program. "Intrcpt time" is the running time when all events relevant to atomicity checking are intercepted but not processed (an empty method is called). For each algorithm, "time" includes the running time of the instrumented program and the subsequent analysis. "space" is the storage used by each algorithm. The storage of the commit-node algorithm is the sum of the number of nodes and the number of inter-edges (which in these experiments is at most 2/3 of the number of nodes) in the trimmed view-forest. The storage of the reduction-based algorithm is the total size of lock-sets (which store identifiers of lock objects) and thread-sets (which store identifiers of thread periods) for all escaped variables. The storage of the block-based algorithm is the number of blocks. "report" reflects the warnings issued by each algorithm. We classify warnings issued by each algorithm into three categories:

Program	LOC	Base time	Intrcpt time	Commit-node Alg			Reduction-based Alg			Block-based Alg		
				time	report	space	time	report	space	time	report	space
elevator	528	0.2s	0.34s	0.4s	0-2-0	342	0.5s	0-2-0	184	0.6s	0-2-0	108
tsp	706	0.24s	0.4s	40.0s	0-2-0	3015	32.5s	0-2-0	530	8m59s	0-2-0	13474
sor	251	0.47s	47.1s	52.4s	0-0-0	90	53.3s	0-0-0	64	1m4.1s	0-0-0	3056
hedc	2197	0.6s	0.82s	1.0s	0-0-0	892	1.0s	0-0-1	349	2.1s	0-0-0	1085
moldyn	1265	44.03s	24m34s	34m26s	0-0-0	3819	38m22.1s	0-0-0	810	28m54.6s	0-0-0	132
montecarlo	3619	15.85s	7m37s	7m43s	0-0-0	148	8m10.1	0-0-0	79	8m11.4s	0-0-0	159
raytracer	1832	14.34s	10m8s	10m50s	2-0-0	106	11m58.9s	2-0-0	26	36m17.6s	2-0-0	39
jigsaw	25012	1.60s	2.2s	3.4s	1-3-0	4031	2.74s	1-3-1	2012	8m25.4s	1-3-0	17254
StringBuffer	1255	-	-	-	0-1-0	-	-	0-1-0	-	-	0-1-0	-
Vector	1020	-	-	-	4-4-0	-	-	4-4-10	-	-	4-4-0	-
Hashtable	1054	-	-	-	0-4-0	-	-	0-4-1	-	-	0-4-0	-
Stack	119	-	-	-	3-4-0	-	-	3-4-12	-	-	3-4-0	-

Table 1. Performance and Accuracy. The categories of "report" for the three algorithms are bug - benign - false alarm. A dash means that the item is negligible.

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

Table 1 shows, for each category, the number of methods such that a warning in that category is issued for a transaction that is an execution of that method or part of the code of that method. For the commit-node algorithm, we count based on the transactions that do not satisfy the condition in the hypothesis of Theorem 4.6.

We conclude the following from Table 1. (1) The commit-node algorithm has the same accuracy on these benchmarks as the pairwise block-based algorithm, and they are more accurate than the reduction-based algorithm; specifically, they produce no false alarms, while the reduction-based algorithm produces 25 false alarms in total. Diagnosing a warning as a false alarm can require significant human time and effort, so reducing the number of false alarms is crucial in practice. (2) In the experiments, the pairwise block-based algorithm does not miss any atomicity violations involving three or more transactions (*i.e.*, no such violations are present). (3) The commit-node algorithm is as fast as the reduction-based algorithm (even 0.4% faster on average), and significantly faster than the block-based algorithm (56% faster on average).

We also check these programs for conflict-atomicity using the commit-node algorithm presented in Section 4.1. It issues exactly the same warnings (including bugs, benign and false alarms) as the commit-node algorithm for checking view-atomicity. This shows that the reduction-based algorithm issues false alarms because its analysis is imprecise, not because it is checking conflict-atomicity while the other algorithms used for Table 1 check view-atomicity. The commit-node algorithm for conflict-atomicity is slightly faster (5.9% faster on average) than the commit-node algorithm for view-atomicity, because the former needs less time to construct inter-edges.

We also test the programs by comparing pair of transactions each time according to Theorems 4.2 and 4.5. Checking pairs of transaction for conflict-atomicity and view-atomicity produces the same result as checking the whole set of transactions.

The bugs in `raytracer` come from atomicity violations involving the field `JGFRayTracerBench.checksum1`, which could get an incorrect value, causing the program to report failure. The bug in `jigsaw` is due to atomicity violations involving the field `w3c.tools.resources.store.ResourceStoreManager.loadedStore` due to statements `loadedStore++` and `loadedStore--` without synchronization; as a result, `loadedStore` may contain an incorrect value. The error in `jigsaw` described in [20] does not appear in our experiments, because the relevant code was modified in the newer version of `jigsaw` that we tested. The above atomicity violations involve data races. The errors in `Vector` and `Stack` are from atomicity violations involving the field `elementCount` (discussed in Section 1).

The reduction-based algorithm produces more false alarms than the others. For example, some `Collection` classes use `modCount` to count modifications. Thus, when an update method m_1 executes `modCount++` (which is a read followed by a write), and another method m_2 checks for recent modifications by reading `modCount`, there is a serializable sequence of events m_1:read(`modCount`) m_2:read(`modCount`) m_1:write(`modCount`). But the benign race on `modCount` causes the reduction-based algorithm to produce a false alarm here, because m_1 contains two accesses to `modCount` that are non-movers. Similar scenarios exist in `jigsaw` (*e.g.*, on the field `alive` in the method `w3c.util.CachedThread.waitForRunner()`) and other programs.

9. Related Work

In [21, 22], we proposed the reduction-based and block-based algorithms for runtime atomicity checking. Flanagan and Freund [5] independently proposed a reduction-based algorithm. Our previous experiments showed that the reduction-based algorithm is faster, and the block-based algorithm is more accurate [22]. This paper presents a new algorithm that is as fast as the reduction-based algorithm, is as accurate as the pairwise block-based algorithm, and can detect atomicity violations involving any number of transactions. We explored the use of static analysis to decrease the overhead for the reduction-based algorithm [17] and the block-based algorithm [1]. The similar technique can be used to reduce the overhead of the commit-node algorithm as our future work.

Flanagan *et. al.* extended their atomicity type system to verify abstract atomicity of programs by analyzing purity [7]. We extended their work to verify atomicity of programs that use non-blocking synchronization [24].

Model checking can also be used to check atomicity [9, 4]. Model checking provides stronger guarantees than runtime monitoring, because it considers all possible behaviors of a program. Also, many of the supporting analyses, such as dynamic escape

analysis, analysis of arrays, deadlock detection, and special treatment of thread-local and read-only variables, etc., can be performed more easily and precisely in a model checker than by program instrumentation [9]. However, model checking is more expensive and is feasible only for programs with relatively small state spaces.

von Praun and Gross [20] present a static analysis to detect violations of *method consistency*, which is similar to atomicity. Although their static analysis is unsound (in order to reduce the cost and the number of false alarms), it considers the entire program and therefore may be more thorough than runtime analysis in some cases. On the other hand, it produces more false alarms than our commit-node algorithm, based on a comparison of the false alarms in our Table 1 with the false and spurious reports in Table 1 of [20].

Linearizability [11] is a correctness condition for objects which are shared by concurrent processes. Linearizability can be viewed as a special case of strict serializability where transactions are restricted to consist of a single method applied to a single object [11]. Linearizability is defined semantically, *i.e.*, in terms of the specification (correctness requirements) of the object. In contrast, we define atomicity in terms of operations performed by the implementation. Our definition is more restrictive but has the practical benefit of being directly applicable to programs for which formal correctness requirements are unavailable. Vafeiadis *et. al.* present an approach to use rely-guarantee reasoning to verify linearizability of several algorithms using fine-grain synchronization [18]. The approach is not automatic but provides static guarantees and can analyze fine-grain synchronization for which our algorithms produce false alarms.

10. Conclusions and Future Work

This paper defines two kinds of atomicity, conflict-atomicity and view-atomicity. In theory, view-atomicity is more appealing because it is less restrictive, but in our experiments, checking view-atomicity and checking conflict-atomicity give the same results. It is well-known that checking conflict-serializability is in P [3] and checking view-serializability is NP-complete [16]; surprisingly, we show that the problems of checking conflict-atomicity and checking view-atomicity are polynomially reducible to each other.

In our experiments, the commit-node algorithms proposed in this paper are as fast as and significantly more accurate than our previous reduction-based algorithm, and they are as accurate as and significantly faster than our previous pairwise block-based algorithm.

Directions for future work include using static analysis to reduce the overhead of the commit-node algorithms, evaluating them on larger applications, and considering fine-grain synchronization.

References

[1] R. Agarwal, A. Sasturkar, L. Wang, and S. D. Stoller. Optimized runtime race detection and atomicity checking using partial discovered types. In *Proc. 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM Press, Nov. 2005.

[2] R. Agarwal, L. Wang, and S. D. Stoller. Detecting potential deadlocks with static analysis and runtime monitoring. In *Proceedings of the Parallel and Distributed Systems: Testing and Debugging (PADTAD) Track of the 2005 IBM Verification Conference*. Springer-Verlag, Nov. 2005.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems.* Addison Wesley, 1987.

[4] C. Flanagan. Verifying commit-atomicity using model-checking. In *Proc. 11th Int'l. SPIN Workshop on Model Checking of Software*, volume 2989 of *LNCS*, pages 252–266. Springer-Verlag, 2004.

[5] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proc. of ACM Symposium on Principles of Programming Languages (POPL)*, pages 256–267. ACM Press, 2004.

[6] C. Flanagan and S. N. Freund. Type inference against races. In *Static Analysis Symposium (SAS)*, volume 3148 of *LNCS*. Springer-Verlag, Aug. 2004.

[7] C. Flanagan, S. N. Freund, and S. Qadeer. Exploiting purity for atomicity. *IEEE Transactions on Software Engineering*, 31(4), Apr. 2005.

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

[9] J. Hatcliff, Robby, and M. B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In *Proc. 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 2937 of *LNCS*. Springer-Verlag, Jan. 2004.

[10] K. Havelund. Using runtime analysis to guide model checking of Java programs. In *Proc. 7th Int'l. SPIN Workshop on Model Checking of Software*, volume 1885 of *LNCS*, pages 245–264. Springer-Verlag, Aug. 2000.

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

[12] Java Grande Forum. Java Grande Multi-threaded Benchmark Suite. version 1.0. Available from http://www.javagrande.org/.

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

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

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

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

[17] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller. Automated type-based analysis of data races and atomicity. In *Proc. ACM SIGPLAN 2005 Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM Press, June 2005.

[18] V. Vefeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearizable objects. In *Proc. ACM SIGPLAN 2006 Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM Press, 2006.

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

[20] C. von Praun and T. R. Gross. Static detection of atomicity violations in object-oriented programs. In *Journal of Object Technology, vol.3, no. 6*, June 2004.

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

[22] L. Wang and S. D. Stoller. Runtime analysis of atomicity for multithreaded programs. Technical Report DAR-04-14, SUNY at Stony Brook, Computer Science Dept., July 2004. (revised May 2005). To appear in *IEEE Transactions on Software Engineering*.

[23] L. Wang and S. D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. Technical Report DAR-05-26, SUNY at Stony Brook, Computer Science Dept., Sept. 2005.

[24] L. Wang and S. D. Stoller. Static analysis for programs with non-blocking synchronization. In *Proc. ACM SIGPLAN 2005 Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM Press, June 2005.