Marathon: Detecting Atomic-Set Serializability Violations with Conflict Graphs

William N. Sumner[1], Christian Hammer[1,2], and Julian Dolby[3]

[1]Purdue University, West Lafayette, IN, USA
wsumner@cs.purdue.edu
[2]Utah State University, Logan, UT, USA
hammer@usu.edu
[3]IBM T. J. Watson Research Center Hawthorne, NY, USA
dolby@us.ibm.com

Abstract. Recent research has proposed several analyses to mitigate the fact that finding concurrency bugs in multi-threaded software is notoriously hard. This work proposes a new analysis based on a correctness criterion called "atomic-set serializability", which incorporates both race conditions and traditional atomicity/serializability. We present a novel analysis based on conflict cycle detection that is guaranteed to find all violations in the intercepted execution trace. A set of heuristics automatically determines all annotations required for atomic-set serializability. We implemented the analysis and evaluated it on a suite consisting of real programs and benchmarks. The evaluation demonstrates the usefulness of our heuristics by finding a number of known (as well as new) violations with competitive overhead and a very low false positive rate.

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

1 Introduction

Multi-threaded programs have become more and more predominant as processor speeds cease to rise significantly, and manufacturers put multiple cores onto one processor. However, writing correct multi-threaded code is notoriously hard, which gave rise to several analyses that statically or dynamically enforce certain correctness criteria. These criteria range from the weakest form, data races on single memory locations, to atomicity for all memory involved in a given transaction. Data races occur when two threads access the same shared variable without synchronization, where one of the accesses is a write. Yet in general, data-race freedom does not guarantee the absence of concurrency-related bugs [1, 2, 7]. A remedy has been found in various definitions of serializability (or atomicity) [13, 23, 33, 38, 39]. According to these definitions, an execution 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. However, serializability/atomicity ignores invariants and consistency

properties that may exist between shared memory locations, and therefore may not accurately reflect the intentions of the programmer for correct behavior, resulting in missed errors and false positives.

A more flexible correctness criterion that takes such relationships into account has been explored recently: *Atomic-set serializability* defines *atomic sets* of memory locations related by some correctness constraint. It further defines *units of work*, operations that preserve these invariants. Since the sets can range from a single location to the entire heap atomic-set serializability subsumes low level data races as well as atomicity [36]. Like serializability, atomic-set serializability disallows concurrency-related errors [1, 2, 7], but it also permits certain non-problematic interleaving scenarios. Atomic-set serializability is based on a declarative specification about data, which can be checked independent from the actual synchronization code, permitting the code to be checked against the programmer's intention, in particular it can be checked independently of specific synchronization constructs such as locks. Therefore, it can be used in settings where many existing approaches cannot, such as classes from the Java 5 `java.util.concurrent` library and lock-free algorithms.

To detect concurrency errors, the intent of the programmer must still be known in terms of the atomic sets of related locations and their corresponding units of work. Declaring them explicitly could impose a significant burden; hence, we explore whether they can be inferred using heuristics based on the assumption that object-oriented code associates units of consistency with objects. We present a set of heuristics (Sect. 4.1) and show that they generate very few false positives (between 2–4%) in terms of our best manual understanding of what the evaluated programs are meant to do.

This work presents a new approach for checking atomic-set serializability based on cycle detection in conflict graphs. The new approach is guaranteed not to miss errors in a given execution with respect to the given atomic sets and units of work, while providing all advantages of atomic-set serializability over previous correctness criteria. Key steps of our technique include:

- Using a simple static escape analysis to detect fields of objects that may be accessed by multiple threads,
- Encoding the dynamic call stack of each thread efficiently [35] based on a static approximation of the call graph,
- Maintaining a conflict graph of units of work in order to detect cycles during execution, which indicates a serializability violation.

Note that all static analyses are for optimization purposes only, our analysis is independent of these preprocessing steps. We implemented the analysis using the *Shrike* bytecode instrumentation component of the WALA program analysis infrastructure. Our tool instruments the bytecodes of an application in order to: (i) intercept accesses to shared data, (ii) maintain a dynamic call graph [35] to determine the units of work to which these accesses belong, and (iii) update the conflict graph accordingly. To encourage problematic interleavings, we optionally instrumented the code with yields, a technique also known as *noise making* [3]. To determine the units of work 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 our tool on a number of benchmarks, including classes from the Java Collections Framework, and applications from the ConTest suite [11]. We found a significant number of violations, including known problems [11, 13], as well as problems not previously reported. Our technique does not miss errors in a given execution, provided our heuristics determine the atomic sets and units of work appropriately. On average over all benchmarks, the instrumentation inserted by our tool slows down program execution by a factor of 4, which is similar to, or better than, the performance overhead incurred by other dynamic serializability violation detection tools [13, 14, 19, 23, 30, 38–40]

In summary, this paper makes the following contributions:

1. We present a dynamic analysis guaranteed to detect all atomic-set serializability violations in the intercepted execution trace based on discovering cycles in a conflict graph. This graph is based on atomic sets and units of work, rather than low-level memory and locking operations in prior work.
2. We incorporated an efficient dynamic call graph encoding scheme that computes the callstack as a small number of integers, and still encompasses all the complexities of object-oriented systems such as exceptions. This uses both less time and less space than traditional approaches.
3. We model the semantics of Object.wait in the context of atomic sets, which leads to a drastic reduction of the false positive rate.
4. We present a set of heuristics that automatically determine the atomic sets and units of work of an application. We demonstrate the usefulness of these heuristics by using them to find many known races and simultaneously keeping the set of false positives very low (2–4%).
5. We implemented this analysis using the WALA infrastructure and show its effectiveness on a number of Java benchmarks. We found known bugs as well as bugs not detected by our previous approach.

2 Background

Our work is based on atomic-set serializability, a correctness criterion for concurrent programs defined by Vaziri et al. [36] which exploits that invariants typically exist between *specific* memory locations; a well-encapsulated data structure will have operations that update only its own memory locations. Atomic-set serializability assumes the existence of *atomic sets* of memory locations that must be updated atomically, and *units of work*, code fragments that preserve consistency of the atomic set, when executed sequentially. Intuitively, the atomic set denotes the elements of a specific data structure, and units of work are the operations for manipulating that data structure.

For cases where an operation needs to happen across multiple data structures, the language offers two more keywords. A parameter declared unitfor signifies that the method is a unit of work for that parameter, and hence this method

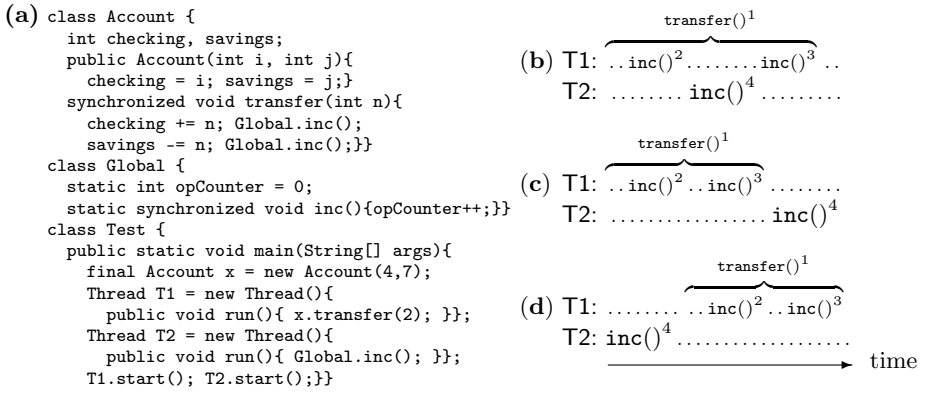

Fig. 1. **(a)** Example program. **(b)**–**(d)** Three different thread executions.

must appear atomic with respect to units of work upon that parameter. For example, the ArrayList constructor from the JDK 1.5.0.18 takes another collection c as parameter without synchronizing on it. Thus, another thread could add or remove elements to c between retrieving the size of c and copying the elements of c to the ArrayList, which results in an inconsistent value of the ArrayLists size. Declaring c `unitfor` expresses the consistency requirement between the two calls.

The owned keyword conceptually declares that a given field is "part of" its containing object by merging the respective atomic sets; this allows composition of more-complex data structures from simpler ones. For example, in the Java Collections, a HashSet is implemented with a backing HashMap stored in a field called map that would be declared `owned` to express the invariant between the state of the set itself and the backing map.

2.1 Example

Figure 1(a) shows a class Account that declares fields checking and savings, as well as a method to transfer money from one to the other. Also shown is a class Global declaring 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. The class Test creates two threads that execute Account.transfer() and Global.inc() concurrently.

Figure 1(b)-(d) depicts executions in which two threads, T1 and T2, concurrently execute the transfer() and inc() methods, respectively. For convenience, each method execution is labeled with a distinct number (1 through 4). Observe that, in Figure 1(b), the execution of inc() by T2 occurs interleaved between that of the two calls to inc() by T1.

2.2 Atomicity/Serializability

For brevity, we only describe these notions on a high level. For a more detailed comparison and the details concerning the example in Figure 1 the reader is referred to our previous work [19].

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 our example, 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 [22] defines a pattern of operations that can be reduced to an equivalent serial execution. However, method transfer() does not correspond to this pattern, so the theory cannot show that no intermediate states are exposed to other threads.

View-serializability. Two executions are *view-equivalent* [4,38] if they contain the same events, each read operation reads the result of the same write operation in both executions, and both executions 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 neither view-equivalent to serial execution (c), nor to serial execution (d). Hence, execution (b) is not view-serializable.

Conflict-serializability. Two events that are executed by different threads are *conflicting* if they operate on the same location and one of them is a write. Two executions are *conflict-equivalent* [4, 38] iff they contain the same events, and each pair of conflicting events appears in the same order. An execution is *conflict-serializable* iff it is conflict-equivalent to a serial execution. Conflict-serializability implies view-serializability [4, 38] as they only differ on how they treat *blind writes*. Hence, execution (b) is not conflict-serializable.

2.3 Atomic-Set Serializability

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, and Global.inc()^4 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 is trivially serial, because events from only one thread are included. Furthermore, the projection onto atomic set S_2 is also serial because the events of units of work Global.inc()^2, Global.inc()^3, and 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 than the traditional notions of atomicity, conflict- and view-serializability. In practice, those would classify execution (b) as having a bug,

(a)

```
                                    x=3; y=2; z=1
                                    fork;
//Unit of Work u1 (T1):  //Unit of Work u2 (T2):  //Unit of Work u3 (T3):
x = 4; y = 3             z = x                    print(z, y)
```

(b) $W_{u1}(x), R_{u2}(x), W_{u2}(z), R_{u3}(z), R_{u3}(y), W_{u1}(y)$

Fig. 2. **(a)** Example threads. **(b)** Non-serializable execution

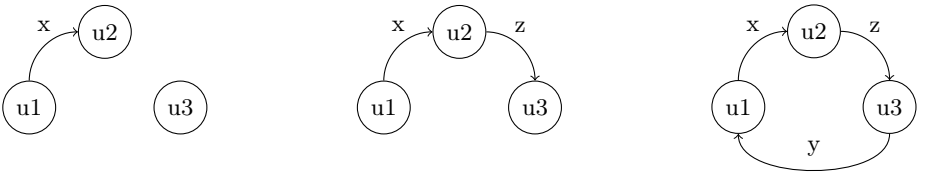

Fig. 3. Conflict graph development for Fig. 2(b) showing a serializability violation as a cycle of conflicts on variables x, y and z between the units of work $u1$, $u2$, and $u3$.

but atomic-set serializability correctly reveals that there is none. Yet, if a coarser granularity of data is desired, all three locations can be placed in a single atomic set, in which case our method reverts to conflict-serializability.

2.4 Overview of Our Approach

The goal of this work is to check atomic-set serializability violations dynamically during program execution. To that end our technique leverages a data structure from database theory called a *conflict graph*. A conflict graph consists of nodes representing the units of work (transactions), and edges modeling conflicts between those. Intuitively, a conflict between two nodes occurs when both units of work access a memory location in an associated atomic set, where one access is a write (see Sect. 3 for formal definitions.) The theory asserts that an execution is serializable if and only if the conflict graph is acyclic.

As an example, consider Fig. 2(a), taken from Wang and Stoller [39, Sect. 6.3], which displays a serializability violation involving 3 threads in part (b). Looking at Fig. 3 reveals the nature of this serializability violation: The execution of Fig. 2(b) induces a conflict graph involving three threads in units $u1, u2$ and $u3$ and the conflict edges are labeled with all three variables involved, so reasoning about this bug is very natural.

3 Algorithm

This section presents the theory behind our new algorithm based on the definition of atomic-set serializability.

Let $\mathcal{L}$ be the set of all memory locations. A subset $L \subseteq \mathcal{L}$ is an *atomic set,* indicating that there *exists* a consistency property between those locations. 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 $loc(e)$ 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 $\mathcal{U}$ be the set of all units of work. We write $sets(u)$ for the set of atomic sets corresponding to u. We say that $\bigcup_{L \in sets(u)} L$ is the *dynamic atomic set* of u. Units of work may be nested, and we write $u \leftarrow u'$ to indicate that u' is nested in u. Units of work form a forest via the $\leftarrow$ relation.

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

A *thread* is a sequence of units of work. The notation $thread(u)$ 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.

Definition 1 (Atomic-set serializability [36]). *An execution is called* atomic-set serializable *if its projections on each atomic set are serializable.*

Definition 2 (Conflict). *Let L be an atomic set, $l \in L$, and u and u' be two units of work for L. Unit u conflicts with u' ($u \rightsquigarrow u'$) if and only if both u and u' access l, at least one of these accesses is a write, and the access in u either reads from or performs the first write of l temporally preceding the access in u'.*

A *conflict graph* is a directed graph where the vertices are the units of work, $\mathcal{U}$, and there exists an edge from u to u' if and only if $u \rightsquigarrow u'$. Figure 3 depict the development of the conflict graph for the executions in Fig. 2(b), with the code shown in part (a).

Lemma 1. *An execution is atomic-set serializable iff its conflict graph is acyclic*

Proof. Follows from our definition of conflict together with previous serializability results [5, 12, 15, 28].

Corollary 1 ([Serializability Violation). *An execution has an* atomic-set serializability violation *iff there exists a cycle in the conflict graph of the execution.*

According to this corollary, the cycles in the conflict graphs of Fig. 3 establish an atomic-set serializability violation in the respective executions of Fig. 2(b).

4 Implementation

This section presents details of our implementation. We first present our choice of defaults for atomic sets and units of work (Sect. 4.1). We then discuss how we perform instrumentation to capture events (Sect. 4.2).

4.1 Automatic Detection of Atomic Sets and Units of Work

We assume that all (including inherited) non-final, non-volatile instance fields of an object are members of an atomic set. All accessible non-static public and protected methods of that object are considered initial units of work declared on this atomic set. 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.

In order to satisfy that each access to an atomic set is done within a corresponding unit of work [36, Sect. 4.1], we assume that a method containing a direct access to a field (or using a simple getter/setter function) is an additional unit of work for the atomic set the field belongs to. A unit of work declared on multiple atomic sets must be a unit of work on their union. Therefore, we merge the original atomic set and the set accessed directly during the execution of the additional unit of work. We support two modi for merging atomic sets: When the direct access is accessing a field of a member of the atomic set, we assume that field is *owned*, so we merge the current atomic set with the one of that member field and propagate that atomic set to the top-most unit of work (see Sect. 3). For direct access to any other field, we do not propagate to the top-most unit, as we assume *unitfor* semantics. Our previous work supported only the owned semantics, which may result in more false positives [19].

Apart from that, we model inner classes. Inner classes indirectly leak access to fields of an enclosing class. For example, in Java Collections, Iterators expose access to an ArrayList's internals to a caller of the iterator() method. Thus, we make the caller a *unitfor* the the enclosing ArrayList as well, protecting access to its internal fields.

These heuristics have been found very effective. They deal correctly with a huge number of access patterns in Java programs.Therefore, we did not add any manual annotations to the programs. We also implemented an intra-procedural static analysis that determines whether a method call is a simple getter/setter.

Modeling wait/notify A call to a.wait() releases the lock associated with a and waits for another thread to signal a certain condition (usually involving a's atomic set). The other thread changes shared state and calls notify(All). When the first thread resumes, it re-evaluates the condition, which would lead to a benign cycle in the conflict graph with a naïve heuristics of units of work. We break a unit of work into two at a call to wait() due to its non-atomic semantics, which is essential for a low false positive rate as shown by our experiments.

Discussion These heuristics are designed to discover atomic sets that cover individual data structures; for many applications, such as building concurrent libraries, this is precisely what is required; however, it is certainly possible to have atomicity violations across data structures. Such races imply dependences between memory locations across data structures that are not isolated behind abstraction boundaries. This suggests a severe breakage of modularity, of which atomicity violations are merely one of many deleterious consequences. Much

work, e.g. alias control such as ownership types [10], has focused on helping programmers eliminate such errors.

4.2 Program Instrumentation

We instrument the program to intercept field access 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] For all benchmarks other than those testing the collections, we did *not* instrument the Java library. All inter-procedural static analyses are purely optimizations to reduce runtime overhead, and we have fallback mechanisms if these analyses fail to complete.

Before instrumentation, our tool performs a simple static escape analysis that 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, as well as access to arrays.

Our tool uses a non-blocking queue similar to [17, Sect. 15.4.2] to store and serialize events of different threads, keeping the *probe effect* [16] (i.e., changes to the system behavior due to observation) as low as possible, and as, under contention, blocking will show degraded performance due to context-switching overhead and scheduling delays. Serializing events in a sequential order is a prerequisite for detecting cycles in the conflict graph. As a field access and its recording do not happen atomically, the scheduler could activate another thread in-between. Nevertheless, the obtained execution is always a valid execution of the program, as the recording takes place in the same thread, and 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, i.e., it might happen with a possible scheduling.

To determine in which unit of work each access belongs, we keep track of a *dynamic call graph*, essentially a call stack, for each called method. An access to a location in an atomic set belongs to its top-most unit of work. To maintain the dynamic call graph, we exploit a technique from Sumner et al. that uses simple arithmetic operations at the invocation points in the program [35]. A call stack corresponds to a path in the static call graph. Using static analysis, we number paths in the call graph and then compute the number of the current path at runtime through addition and subtraction. To handle callbacks and recursion, we represent the dynamic call graph as a list of numbers, saving the last computed id to the list before such a callback and restoring it from the list afterward. We further extend the technique to handle exceptions in Java by saving the id before a try and restoring it within a catch or finally.

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 [3].

[1] http://wala.sf.net

To reduce the memory overhead of our technique, we additionally garbage collect old units of work that can no longer lead to cycles in the conflict graph. When a unit of work completes and has no incoming conflict edges, it cannot participate in a conflict and may be safely collected. This further allows any terminated units of work conflicting only with the collected one to also be collected.

5 Evaluation

We evaluated our new analysis on the same set of benchmarks as the previous analysis [19] and additional real world programs, including ConTest [11], Java Collections, the Jigsaw webserver, and the Jspider and Weblech web crawlers from [29]. We ran all benchmarks on a 64-bit 2.8GHz 4-core Intel machine with 6GB memory and used the Sun Hotspot JVM version 1.6.0_24-b07.

Table 1 shows the results of our analysis, where each benchmark was executed twice. The column "Program" lists the name of each benchmark. We first list benchmarks from the ConTest suite and then other benchmarks. The "LOC" column contains the number of static lines of code. While the ConTest benchmarks are small kernel programs, others range from a few thousand to more than 100K LOC, showing the applicability of the technique to real world programs. The "#Threads" column lists the configured number of threads in the benchmark. For the ConTest and Collections benchmarks, these are the same as in previous work.

We evaluated the reported violations along several dimensions: The *unique* cycles are counted in the "Cycle Sizes" column according to the number of units of work that the cycle comprises. When multiple accesses (on possibly different fields of an atomic set) can induce the same cycle in the conflict graph, they are considered parts of the same violation and only counted once. However, we listed cycles involving the same atomic set with different sizes separately.

The column "FP" displays the ratio of benign violations (false positives) and the total violations reported based on manual inspection of the programmer's intentions. With our model of Object.wait, only few violations did not in fact indicate a bug. For example, the programs BufWriter, Lottery and Manager had cases where our heuristics for units of work were too coarse grained. Exploring further options for splitting up units of work at certain places like thread fork or join points is subject to future work. Overall, our false positive rate is just under 2%. The Collections benchmarks are admittedly pathological in the number of violations they observe, but even excluding them, our false positive rate is 4%. It was interesting to see the number of violations found for Piper reduce from 75 to 0 when we introduced splitting of the unit of work at Object.wait callsites (see Sect. 4.1). Also BoundedBuffer, JSpider and Weblech would have had several false positives without splitting. These numbers show that faithfully modeling the semantics of wait reduces the false positive rate considerably.

The "New Vio" column compares the current technique with our previous approach [19]. It lists the number of new serializability violations detected. Pro-

Table 1. For each benchmark, the table indicates the number of different violations detected by cycle length, false positives, new violations, slowdown factor, max. size of the conflict graph, and the avg. call stack size

Program	LOC	Cycle Sizes								FP	New	SF	SF	$\|CG\|$	Stack	#Threads
		2	3	4	5	6	7	8	9+		Vio	Mem	Disk		Depth	
Account	155	1	0	0	0	0	0	0	0	0/1	0	1.0	1.0	14	2.0	10
AirlineTickets	95	1	1	0	0	0	0	0	0	0/2	0	1.5	1.5	94	2.0	100
AllocationV	286	0	0	0	0	0	0	0	0	0/0	0	1.0	1.0	4	2.0	2
BoundedBuffer	328	0	0	1	0	0	0	0	0	0/1	–	1.0	1.0	10	2.0	3
BubbleSort	362	2	3	1	0	0	0	0	0	0/6	0	1.0	1.0	17	2.0	8
BubbleSort 2	130	1	1	1	1	1	1	1	27	0/34	34	10.4	1.4	201	2.0	200
BufWriter	255	1	2	0	0	0	0	0	0	2/3	2	–	–	8	2.0	6
Critical	68	1	0	0	0	0	0	0	0	0/1	0	1.0	1.0	2	2.0	2
DCL	183	1	1	1	0	0	0	0	0	0/1	0	1.3	1.3	31	2.0	20
FileWriter	325	0	0	0	0	0	0	0	0	0/0	0	1.0	1.0	6	2.0	N/A
LinkedList	416	1	0	0	0	0	0	0	0	0/1	0	1.0	1.0	9	2.0	2
Lottery	359	2	1	1	1	1	0	0	0	1/6	5	1.5	1.5	98	2.0	33
Manager	188	4	0	0	0	0	0	0	0	2/4	2	1.0	1.0	7	2.0	3
MergeSort	375	1	0	0	0	0	0	0	0	1/1	0	1.1	1.1	12	3.4	4
MergeSortBug	257	2	1	2	1	1	0	0	0	0/7	1	8.7	1.1	27	3.6	4
PingPong	272	1	0	0	0	0	0	0	0	0/1	0	1.0	1.0	124	2.0	120
Piper	116	0	0	0	0	0	0	0	0	0/0	–	–	–	83	2.0	40
ProducerConsumer	223	3	3	0	0	0	0	0	0	0/6	–	1.0	1.0	12	2.0	6
Shop	273	2	1	1	1	1	1	1	17	0/25	1	1.0	1.0	122	2.0	7
SunsAccount	144	2	1	1	1	1	1	1	31	0/39	1	1.0	1.0	7613	2.0	N/A
Jigsaw	142K	1	0	0	0	0	0	0	0	0/1	0	3.9	3.9	94	8.0	3
Jspider	56K	4	0	0	0	0	0	0	0	0/4	–	1.2	1.2	128	3.4	6
Weblech	1874	2	0	0	0	0	0	0	0	0/2	–	1.1	1.0	12	2.0	9
ArrayBlockingQ	1576	1	1	0	0	0	0	0	0	0/7	2	26.6	14.12	725	4.1	10
ArrayList (sync)	2266	24	37	10	5	3	0	0	0	0/60	60	48.9	19.6	429	4.1	10
LinkedBlockingQ	1620	1	0	0	0	0	0	0	0	0/1	1	20.1	16.9	605	4.0	10
DelayQueue	1961	25	13	3	1	1	0	0	0	0/43	20	23	17.5	155	4.2	10
Vector	2636	18	38	36	24	11	4	0	0	0/131	131	52.8	10.4	63	4.0	10

grams not previously evaluated are denoted by –. We interpret the substantial number of new violations as an indication of the benefit over the old technique, stemming from the fact that the new technique does not miss violations in the intercepted execution. We note that the newly found violations also mean that our technique has no false negatives with respect to the known bugs in the ConTest and Collections benchmarks except for in AllocationV, FileWriter, and MergeSort. In these benchmarks, poor object orientation as discussed in Sect. 4 and an inability to reproduce a failing run prevented us from detecting violations.

The "SF" columns indicate the slowdown factor of the instrumented version compared to the uninstrumented version of the program. "SF Mem" is the slowdown when the conflict graph is maintained online during program execution, and "SF Disk" is the slowdown when all accesses are logged to disk and cycle detection is performed postmortem. Note that Piper exhibits a bug that prevented it from terminating and being timed, and BufWriter terminates its threads predictably after 10 seconds. We excluded these from the average, denoted by –. For Jigsaw, we measured the slowdown in response time for client requests, as a web server runs in an infinite loop. Our technique is comparable or better than previous approaches, which range from 10x-200x [13, 14, 19, 23, 30, 38–40], having a 8.5x average overhead factor when performed online. This, however, is biased by the pathological Collection benchmarks. When only the programs with

more realistic behavior are considered, the overhead diminishes to 1.9x. When cycle detection is performed postmortem, these numbers diminish to 4x and 1.1x respectively. Some benchmarks, in particular our synthetic test harness for the Collections exhibit pathological behavior such that every field access must be checked for potential conflicts, resulting in atypical overhead. We include the Collections data to show that out technique works even on degenerate programs. In particular, most techniques checking atomicity or serializability violations depend on a particular locking discipline and are thus not suitable for the highly concurrent data structures of the package java.util.concurrent.

Taken from our cycle detection algorithm [18], our technique has a theoretical time complexity of $O(n^{3/2})$ where n is the number of accesses to shared variables. In reality, our technique is efficient and practical because real world programs do not have such degenerate behavior. In practice, conflict graphs do not grow very large, as seen in Table 1. This is the result of the pruning from Sect. 4.2, and it reduces the practical cost of cycle detection. That is, the practical running time of the algorithm is no longer proportional to an execution's length. In addition, the number of variables that escape across multiple threads is limited, ensuring that much of an execution can usually be ignored by the analysis.

The column "$|CG|$" shows the maximum size of the conflict graph before each garbage collection, given as the number of units of work in the graph. "Stack depth" shows the number of integers required for our compact stack encoding. There is no consistent correspondence between these statistics and the apparent slowdown factor, which supports our argument on the algorithm's complexity.

6 Related Work

A data race occurs when there are two concurrent accesses to a shared memory location not ordered by synchronization, at least one of which is a write. Dynamic analyses for detecting data races include those based on the lockset algorithm [32,34], on the happens-before relation [25], or on a combination of the two [27]. Dynamic approaches to detecting races scale reasonably well for real applications and have detected a large number of bugs in real software [27,32,33].

Narayanasamy et al. [26] 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.

A program without data races may not be free of concurrency bugs as shown in [2,7]. Atomic-set serializability captures these forms of high-level data races as a correctness criterion based on the programmer's intentions for correct behavior directly. Unlike these techniques, our approach is independent of any synchronization mechanism.

Atomizer [13] , is a dynamic atomicity checker based on Lipton's theory of reduction. Wang and Stoller present a number of different algorithms for detecting atomicity violations [38,39]. The Block-Based Algorithm [39] is based on

non-serializable interleaving patterns. In addition, 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 [38] present two Commit-Node Algorithms for checking view serializability and conflict serializability (detailed comparison presented in [19]).

Lu et al. [23] 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.

Another serializability violation detector was presented by Xu et al. [40]. 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. Nonserializability checking is done using a heuristic based on strict two-phase locking. Like us, it does not rely on the possibly buggy locking structure of the program.

Recently, Park et al. correlated access patterns with the observed likelihood or suspicion that they cause a program to behave incorrectly [30]. They ignore such problems as stale writes and inconsistent reads, and they do not handle unserializable behaviors between more than two threads.

Other recent work uses cycle detection to find atomicity violations: Farzan et al. use postmortem cycle detection to find atomicity violations requiring user specified transactions [12]. Velodrome dynamically detects atomicity violations [14]. It uses a similar mechanism for safely garbage collecting terminated transactions. Atomicity does not take the consistency properties between data into account and thus may ignore the programmer's intentions as exemplified in Sect. 2.2. While Velodrome's analysis should be both sound and complete, their implementation is neither. This is due to slightly unsound optimizations, and because Velodrome makes the heuristic assumption that all methods are atomic, which is not generally the case, like for Thread's `run()` methods. We argue that our heuristics based on OO principles and the declarative approach to synchronization models the programmer's intentions better. As for false positives, they report none; however, that is with respect to their very strong assumptions. In contrast, our reports of false positives are with respect to the programmer's intentions as measured by potential to produce wrong answers. Finally, Velodrome does not instrument array access for reasons of complexity, which would have resulted in more than 22% missed violations in our ConTest benchmarks.

Related work also explores alternative thread schedules that might cause atomicity violations to occur [6, 8, 9, 14, 20, 21, 31]. We leave this orthogonal problem as future work. The work of Burnim et al. [6] also extends to such difficult data structures as those in `java.util.concurrent`.

Martin et al. [24] propose dynamic ownership policy checking for shared objects in C/C++. Their approach requires manual ownership annotations and imposes an average runtime overhead of 26%. Working on a very fine granularity

level, their annotations could in theory be used to check atomic-set serializability, however, by annotating code instead of data their approach is not data-centric.

A previous approach of Hammer et al. [19] matches an intercepted execution trace against a set of problematic interleaving patterns. Unlike conflict graphs, that approach cannot find all possible atomic-set serializability violations in an intercepted execution trace. Apart from that, that work used a different heuristics to determine units of work and atomic sets. In particular it only supported the owned annotation, did not infer direct field access in accessor methods, needed to retain the exact index of array access for maximal precision and could not optimize away events accessing the same atomic set and unit of work.

7 Conclusions

This work presents a new mechanism to dynamically detect atomic-set serializability violations. It is both more powerful than previous atomic-set serializability violation detectors, for identifying all violations present in the intercepted execution, as well as detectors of other correctness criteria like race freedom, serializability, and atomicity, as these are subsumed by the notion of atomic-set serializability. We have shown that our new algorithm scales to realistic program sizes. We also proposed a set of heuristics to determine atomic sets and units of work and demonstrate their effectiveness in the evaluation where they successfully find many known concurrency bugs with a very low false positive rate. Even though our analysis already finds a high number of violations due to noise making, we envisage prediction of atomic-set serializability violations in alternative schedules of the program as a possible extension, to mitigate coverage of the huge test space of concurrent programs.

Acknowledgement. We are grateful to Frank Tip, Mandana Vaziri, and Pavel Avgustinov for discussions on this approach. This work was supported in part by NSF grant CCF 1048398.

References

1. Artho, C., Havelund, K., Biere, A.: High-level data races. Journal on Software Testing, Verification and Reliability (STVR) 13(4), 207–227 (2003)
2. Artho, C., Havelund, K., Biere, A.: Using Block-local Atomicity to Detect Stale-value Concurrency Errors. In: ATVA 2004.
3. Ben-Asher, Y., Eytani, Y., Farchi, E., Ur, S.: Noise makers need to know where to be silent - producing schedules that find bugs. In: ISOLA 2006.
4. Bernstein, P., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley (1987)
5. Bernstein, P.A., Goodman, N.: Concurrency Control in Distributed Database Systems. ACM Comput. Surv. 13(2), 185–221 (1981)
6. Burnim, J., Necula, G., Sen, K.: Specifying and checking semantic atomicity for multithreaded programs. In: ASPLOS 2011.
7. Burrows, M., Leino, K.R.M.: Finding stale-value errors in concurrent programs. Concurrency and Computation: Practice and Experience 16(12), 1161–1172 (2004)
8. Chen, Q., Wang, L.: An Integrated Framework for Checking Concurrency-Related Programming Errors. In: COMPSAC 2009.
9. Chen, Q., Wang, L., Yang, Z., Stoller, S.: HAVE: Detecting Atomicity Violations via Integrated Dynamic and Static Analysis. In: FASE 2009.

10. Clarke, D.G., Noble, J., Potter, J.M.: Simple Ownership Types for Object Containment, In: ECOOP 2001. LNCS, vol. 2072, pp. 53–76.
11. Eytani, Y., Ur, S.: Compiling a benchmark of documented multi-threaded bugs. In: IPDPS 2004.
12. Farzan, A., Madhusudan, P.: Monitoring Atomicity in Concurrent Programs. In: CAV 2008.
13. Flanagan, C., Freund, S.N.: Atomizer: a dynamic atomicity checker for multithreaded programs. In: POPL 2004
14. Flanagan, C., Freund, S.N., Yi, J.: Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In: PLDI 2008
15. Fle, M.P., Roucairol, G.: On serializability of iterated transactions. In: PODC 1982
16. Gait, J.: A probe effect in concurrent programs. Software: Practice and Experience 16(3), 225–233 (1986)
17. Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., Lea, D.: Java Concurrency in Practice. Addison Wesley Professional (May 2006)
18. Hauepler, B., Kavitha, T., Mathew, R., Sen, S., Tarjan, R.: Faster algorithms for incremental topological ordering. In: ICALP 2008
19. Hammer, C., Dolby, J., Vaziri, M., Tip, F.: Dynamic detection of atomic-set-serializability violations. In: ICSE 2008
20. Kahlon, V., Wang, C.: Universal Causality Graphs: A Precise Happens-Before Model for Detecting Bugs in Concurrent Programs. In: CAV 2010
21. Lai, Z., Cheung, S.C., Chan, W.K.: Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In: ICSE 2010
22. Lipton, R.J.: Reduction: a method of proving properties of parallel programs. Commun. ACM 18(12), 717–721 (1975)
23. Lu, S., Tucek, J., Qin, F., Zhou, Y.: AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In: ASPLOS 2006
24. Martin, J.P., Hicks, M., Costa, M., Akritidis, P., Castro, M.: Dynamically checking ownership policies in concurrent C/C++ programs. In: POPL 2010
25. Min, S.L., Choi, J.D.: An efficient cache-based access anomaly detection scheme. In: ASPLOS '1991
26. Narayanasamy, S., Wang, Z., Tigani, J., Edwards, A., Calder, B.: Automatically classifying benign and harmful data races using replay analysis. In: PLDI 2007
27. O'Callahan, R., Choi, J.D.: Hybrid dynamic data race detection. In: PPoPP 2003
28. Papadimitriou, C.: The theory of database concurrency control. Computer Science Press, Inc., New York, NY, USA (1986)
29. Park, C., Sen, K.: Randomized active atomicity violation detection in concurrent programs. In: FSE 2008
30. Park, S., Vuduc, R.W., Harrold, M.J.: Falcon: fault localization in concurrent programs. In: ICSE 2010
31. Park, S., Lu, S., Zhou, Y.: CTrigger: exposing atomicity violation bugs from their hiding places. In: ASPLOS 2009
32. von Praun, C., Gross, T.R.: Object race detection. In: OOPSLA 2001
33. von Praun, C., Gross, T.R.: Atomicity Violations in Object-Oriented Programs. Journal of Object Technology 3(6), 103–122 (June 2004)
34. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: a dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. 15(4), 391–411 (1997)
35. Sumner, W.N., Zheng, Y., Weeratunge, D., Zhang, X.: Precise calling context encoding. In: ICSE 2010
36. Vaziri, M., Tip, F., Dolby, J.: Associating synchronization constraints with data in an object-oriented language. In: POPL 2006
37. Vaziri, M., Tip, F., Dolby, J., Hammer, C., Vitek, J.: A Type System for Data-Centric Synchronization. In: ECOOP 2010
38. Wang, L., Stoller, S.D.: Accurate and Efficient Runtime Detection of Atomicity Errors in Concurrent Programs. In: PPoPP 2006
39. Wang, L., Stoller, S.D.: Runtime Analysis of Atomicity for Multithreaded Programs. IEEE Transactions on Software Engineering 32(2), 93–110 (2006)
40. Xu, M., Bodík, R., Hill, M.D.: A serializability violation detector for shared-memory server programs. In: PLDI 2005