Specifying and Checking Semantic Atomicity for Multithreaded Programs

Jacob Burnim George Necula Koushik Sen
University of California, Berkeley
{jburnim,necula,ksen}@cs.berkeley.edu

Abstract

In practice, it is quite difficult to write correct multithreaded programs due to the potential for unintended and nondeterministic interference between parallel threads. A fundamental correctness property for such programs is *atomicity*—a block of code in a program is *atomic* if, for any parallel execution of the program, there is an execution with the same overall program behavior in which the block is executed serially.

We propose *semantic atomicity*, a generalization of atomicity with respect to a programmer-defined notion of equivalent behavior. We propose an assertion framework in which a programmer can use *bridge predicates* to specify noninterference properties at the level of abstraction of their application. Further, we propose a novel algorithm for systematically testing atomicity specifications on parallel executions with a bounded number of *interruptions*—i.e. atomic blocks whose execution is interleaved with that of other threads. We further propose a set of sound heuristics and optional user annotations that increase the efficiency of checking atomicity specifications in the common case where the specifications hold.

We have implemented our assertion framework for specifying and checking semantic atomicity for parallel Java programs, and we have written semantic atomicity specifications for a number of benchmarks. We found that using bridge predicates allowed us to specify the natural and intended atomic behavior of a wider range of programs than did previous approaches. Further, in checking our specifications, we found several previously unknown bugs, including in the widely-used `java.util.concurrent` library.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Algorithms, Reliability, Verification

1. Introduction

With the growing prevalence of multicore processors, it is increasingly necessary for programmers to write parallel software. Yet parallel programming remains notoriously difficult. A key reason for this difficulty is that the parallel threads of a multithreaded program may nondeterministically interfere with one another.

Thus, a fundamental parallel correctness property for multithreaded programs is *atomicity* [9]. A block of code is *atomic* if it appears to execute all at once, indivisibly and without interruption from any other program thread. The behavior of an *atomic* code block can be understood and reasoned about *sequentially*, as no parallel operations can interfere with its execution.

Many researchers have proposed using *transactional memory* hardware, libraries, and/or language constructs to implement such atomic blocks. But in much existing multithreaded code, desired atomicity is implemented using a variety of synchronization techniques, including coarse or fine-grained locking and non-blocking synchronization with primitives such as atomic compare-and-swap. Correctly implementing atomicity using these techniques can be difficult and error-prone. Thus, as we discuss in Section 6, there has been great interest in techniques enabling programmers to specify what fragments of their concurrent programs behave as if atomic, and in techniques for testing or verifying that such programs conform to their atomicity specifications.

Traditional notions of atomicity are often too strict in practice, however, because they require the existence of serial executions that result in an state identical to that of the interleaved execution. We propose an assertion framework that allows programmers to specify that their code is *semantically* atomic—that any parallel, interleaved execution of an atomic block will have an effect *semantically equivalent* to that of executing the block serially. Programmers specify this semantic equivalence using *bridge predicates* [3]—predicates relating pairs of program states from the interleaved and the equivalent serial execution. Such predicates allow the equivalence of executions to be defined at the level of abstraction of an application.

We further propose an approach to check our semantic atomicity specifications by testing whether or not specified programs are *semantically linearizable*. We choose to check linearizability because (1) this stronger notion is significantly easier to check since the restriction on allowed serial executions significantly reduces the space of serial executions that we must search, and (2) the notion of linearizability is often used to describe the parallel correctness of various concurrent data structures. Essentially, to test linearizability for a particular interleaving we need to consider only permutations of atomic blocks that overlapped in the interleaved execution.

The key to the efficiency of our approach is based on two observations. First, linearizability can be checked efficiently for a parallel execution in which only a small number of atomic blocks overlap, since we need to examine only a small number of similar sequential executions. Second, our experience shows that most atomicity bugs can be reproduced with a small number of overlapping atomic blocks. Thus, we test linearizability of a program by generating parallel executions with only a small number of interrupted atomic blocks. Our experiments show that we can effec-

ASPLOS'11, March 5–11, 2011, Newport Beach, California, USA.
Copyright © 2011 ACM 978-1-4503-0266-1/11/03. . . $10.00

tively find serious atomicity errors in our benchmarks by testing such *interruption-bounded* executions.

To further reduce the search space for the linearized execution, we propose a set of sound heuristics and optional user annotations. We show in our experiments that in the presence of such annotations we can often find the linearized execution in the first attempt.

We have implemented our assertion framework for Java and used it to specify the intended atomicity of a number of benchmarks. We found that the ability to specify atomicity at the *semantic* level, using bridge predicates, is crucial for capturing the intended atomic behavior of many of our of the benchmarks. Such benchmarks contain sections of code that, while semantically atomic, are not atomic under more strict, traditional notions of atomicity.

In summary, we describe the following contributions:

- We propose using *bridge predicates* to specify that regions of parallel programs are intended to be *semantically atomic*. Our notion of *semantic atomicity* is more general than traditional strict notions of atomicity and is applicable to a wider range of parallel programs.
- We propose an approach to test efficiently and effectively a program's semantic atomicity specification by checking the *linearizability* of program executions with a bounded number of interrupted atomic sections. We further propose program annotations and corresponding heuristics to reduce significantly the search space that must be explored during testing, without sacrificing any ability to find atomicity errors.
- We implement an assertion framework for Java for specifying and testing semantic atomicity specifications and evaluate our approach on a number of Java benchmarks. We find that bridge predicates are required in a majority of the examples. We show that the heuristics we propose make the testing approach both reasonably efficient and effective at finding bugs.
- We find a number of previously unknown atomicity errors, including several in Java's built-in data structure libraries.

2. Specifying Semantic Atomicity

In this section, we informally describe atomicity and motivate our proposal for semantic atomicity specifications. We first describe a real-world motivating example. We then expand on semantic atomicity specifications using two simpler examples. In Section 2.1, we discuss the effort involved in programmers writing such atomicity specifications.

We consider parallel programs in which certain regions of code are annotated as *atomic blocks*. This annotation specifies the programmers *belief* or *intention* that each atomic block is written so that, however the block is actually executed, the effect is *as if* the block's execution occurred all-at-once, with no interference or interruption from other parallel threads. For simplicity, we consider each indivisible program instruction that is not in a user-annotated atomic block to be in its own implicit, single-instruction block.

Example 1: Concurrent Queue Consider the example program in Figure 1 using Java's `ConcurrentLinkedQueue` data structure, an implementation of Michael and Scott's non-blocking queue [23]. `ConcurrentLinkedQueue`, from the `java.util.concurrent` package, is implemented in a lock-free, non-blocking manner, updating its internal structure using compare-and-swap operations. If two parallel queue operations conflict, one of the operations will detect the conflict and retry.

The implementation of `ConcurrentLinkedQueue` is designed to ensure that, when multiple queue operations occur concurrently, their result is the same as if all queue operations had been executed atomically. We specify this intended atomicity in Figure 1 by enclosing each parallel call to `remove(1)` in a specified *atomic block*, which we write as `@assert_atomic {...}`.

```
     Queue q = new ConcurrentLinkedQueue();
     q.add(1);  q.add(1);

   thread1:                   thread2:
    @assert_atomic {           @assert_atomic {
       q.remove(1);               q.remove(1);
    }                          }
           bridge predicate:
                 q.equals(q')
```

Figure 1. Example program with highly-concurrent queue. The queue initially contains two copies of the value 1, and two parallel threads each try to remove a 1 from the queue. These remove operations are specified to execute as if atomic. The program is not strictly atomic, but is *semantically* atomic with respect to the given *bridge predicate*.

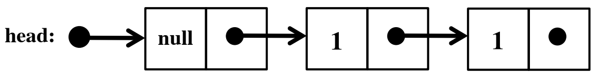

Figure 2. Initial internal structure of queue q.

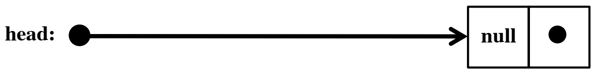

Figure 3. Internal structure of queue q after *any* serial execution of the example program.

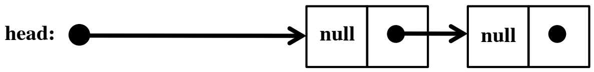

Figure 4. Possible structure of q after a parallel, interleaved execution of the example program.

If our specification was a strict atomicity specification, it would assert: for any parallel execution of the program, in which the calls to `remove(1)` can interleave, there there must exist a *serial* execution, in which each call to `remove(1)` occurred atomically, producing an *identical* final state. But this strict atomicity specification does not hold.

Internally, a `ConcurrentLinkedQueue` is a linked list. Nodes in the list can be lazily deleted—i.e. a remove operation can set the data field of a node to `null`, indicating that the corresponding data element is no longer in the queue, but leave removing the node to some future operation.

Before the parallel threads execute `remove`, the internal list structure is as shown in Figure 2. In any serial execution of the two calls to `remove`, the second call will lazily delete the node `null`'ed by the first call, yielding the final internal queue structure shown in Figure 3. But in a parallel execution in which the two calls to `remove` are interleaved, it is possible for neither call to clean up after the other, yielding the final internal state of the queue shown in Figure 4.

Thus, under a traditional, strict definition of atomicity, the `remove` method is *not* atomic, as a non-serial, interleaved execution can yield a program state not reachable by any serial execution. But in either case the abstract, semantic state of queue q is the same—the queue is empty! That is, the code is atomic, but only at the *semantic* level of the contents of the queue.

```
int balance = 0;


void deposit1(int a) {
  @assert_atomic {
    int t = balance;
    t += a;
    balance = t;
  }
}
```

```
int balance = 0;


void deposit2(int a) {
  @assert_atomic {
    int t = balance;
    while (!CAS(&balance, t, t + a)) {
      t = balance;
    }
  }
}
```

```
int conflicts = 0;
int balance = 0;

void deposit3(int a) {
  @assert_atomic {
    int t = balance;
    while (!CAS(&balance, t, t + a)) {
      conflicts += 1;
      t = balance;
    }
  }
}
      bridge predicate:
         balance == balance'
```

Figure 5. Three different implementations of a function to make a deposit into a bank account. Implementation `deposit1` is not atomic, while `deposit2` *is* atomic. Implementation `deposit3` is not strictly atomic, but it is *semantically* atomic with respect to the bridge predicate `balance==balance'`.

To capture this kind of parallel correctness property, we propose *semantic atomicity*. Blocks of code are semantically atomic when, however their execution is interleaved with that of other code, their effect is *semantically equivalent* to their effect when executed serially. The desired semantic equivalence is specified by a programmer using a *bridge predicate*. For this program, semantic equivalence is given by the bridge predicate $\Phi(\sigma, \sigma')$:

$$\texttt{q.equals}(\texttt{q}')$$

This bridge predicate compares two program states, σ and σ', the first from an interleaved, parallel execution of our example program and the second from a serial execution. The unprimed `q` refers to the queue in state σ and the primed $\texttt{q}'$ refers to the queue in state σ'. This bridge predicate specifies states σ and σ' are semantically equivalent when the `equals` method of `ConcurrentLinkedQueue` returns *true* on the queues from the two states—that is, when the two queues contain the same elements, independent of their internal structure.

This semantic atomicity specification *does hold* for our example program. For any parallel execution of the program, there exists a serial execution that produces an equivalent final queue `q`.

Example 2: Bank Account Consider the code in Figure 5 for function `deposit1` for making a deposit into a bank account whose balance is stored in variable `balance`. The atomic specification in `deposit1` *does not hold* because there is an interleaved execution for which there is no equivalent serial execution. Suppose two threads call `deposit1`(100) in parallel, with `balance` initially 0. Under certain interleavings, both calls to `deposit1` can read a `balance` of 0 and then both can write 100 to `balance`, producing a wrong final result: `balance=100`. In contrast, any serial execution of the two threads, in which the body of `deposit1` cannot be interleaved with any other code, must produce the final result `balance=200`.

```
       List list = new Vector();

    thread1:                 thread2:
     @assert_atomic {         @assert_atomic {
        ...                        ...
        list.add(1);               list.add(3);
        ...                        ...
        list.add(2);               list.add(4);
     }                        }

          bridge predicate:
             equalMultisets(list, list')
```

Figure 6. Example program in which two atomic blocks insert elements into a synchronized list.

Note that atomicity violations can occur even in code that is free of data races. For example, if `deposit1` held a shared lock while reading and writing to `balance`, but released the lock in between when executing "`t += a`", then the procedure would be free of data races but would still not be atomic.

Consider instead the implementation `deposit2` in Figure 5, which uses an atomic compare-and-swap (`CAS`) operation to modify variable `balance`. In this implementation, the atomicity specification *does hold*. Each call to `deposit` reads `balance` into a temporary `t` and then attempts to atomically update `balance` to $\texttt{t} + \texttt{a}$, succeeding if `balance` still equals `t`. If some other thread interferes, changing `balance` between the read of `balance` and the `CAS`, then the atomic update fails and `deposit` re-reads `balance` and tries again.

Finally, consider the implementation `deposit3` in Figure 5. This code is identical to `deposit2`, except that a count is kept, in shared variable `conflicts`, of the approximate[1] number of failed compare-and-swap operations during runs of `deposit3`.

Due to the introduction of shared counter `conflicts`, the atomic specification no longer holds. In a parallel execution with two calls to `deposit3(100)`, if the execution of the methods interleave, it is possible for `conflicts` to be incremented to 1. But in any execution in which the methods are executed serially, the value of `conflicts` will be 0.

However, the `deposit3` implementation is semantically atomic with respect to the bridge predicate:

$$\texttt{balance} == \texttt{balance}'$$

That is, if some interleaved execution produces a final `balance`, there will exist a serial execution producing a final $\texttt{balance}'$ such that `balance` and $\texttt{balance}'$ are equal.

Example 3: Multiset Stored as a List Consider the example program in Figure 6, in which two threads accumulate integers into a shared `list`. If the programmer cares about the exact order of the final elements in `list`, then this code is *not* atomic. Although method `add` of `Vector` is synchronized, an interleaved execution of this example program can yield a final list of `[1,3,2,4]`, while a serial execution of the two atomic blocks could yield only `[1,2,3,4]` or `[3,4,1,2]`.

[1] Such performance counters are often not synchronized, as developers reason that the cost of synchronization is too great and approximate counts are often suitable for performance debugging.

But this code can be thought of as atomic if the programmer cares only about the *multiset* of elements in the final list. That is, the example program in Figure 6 is semantically atomic with respect to some bridge predicate like equalMultisets(list,list'), where equalMultisets is a function that compares two collections to see if they have equal multisets of elements.

2.1 Writing Semantic Atomicity Specs

Our above examples demonstrate that we can reap the benefits of atomicity as a specification and reasoning tool in many more contexts if we consider the more relaxed form of semantic atomicity with respect to an application-specific bridge predicate. But how much effort is involved in writing such a specification for a parallel program?

To write a semantic atomicity specification for a program, a programmer must: (1) Indicate with @assert_atomic which static blocks of code are intended to execute as if atomic, and (2) Write a bridge predicate to define when two final states of the program are semantically equivalent. We believe neither task should be difficult for the author of a parallel program.

We believe that, when writing multithreaded software, programmers must already be thinking about the possible interference between parallel tasks and how to prevent harmful interference using, for example, thread-safe data structures, locks, or atomic primitives such as compare-and-swaps. That is, programmers are already thinking about how to ensure that program tasks are atomic at some semantic level. Thus, it should not be difficult to specify which blocks of code are intended to behave equivalently whether or not other program tasks are run concurrently—for example, a modification to concurrent queue data structure or a deposit to a bank account. Further, in our experimental evaluation (Section 5), we found it to be quite simple to identify the intended atomic blocks in our benchmark programs. We specified between one and eight atomic blocks for each such benchmark.

We similarly believe that it is straightforward to specify when two program results are semantically equivalent using a bridge predicate. This specification task does not require reasoning about possible program interleavings, but simply identifying which variables or objects hold the final result of a program and considering when two such final results are semantically the same—for example, that two queues are equivalent if they contain the same elements in the same order, independent of the structure of their internal linked lists. Further, in our experimental evaluation (Section 5), we found that writing such bridge predicates required only a few lines of specification for each benchmark.

3. Semantic Atomicity and Linearizability

In this section we elaborate on several possible interpretations of atomicity specifications. We first describe these notions at a high level, and compare them to other notions of parallel correctness and non-interference. Later in this section we give the precise formal foundations on which we developed the checking algorithm described in the rest of this paper.

3.1 Overview

At a high-level, we think of a parallel program annotated with atomic blocks as having two different execution semantics: (1) In the *interleaved* or *non-serial* semantics, the atomic annotations have no effect—the parallel operations of different threads can be freely interleaved. (2) In the *serial* or *non-interleaved* semantics, when one program thread enters an atomic block, no other thread may execute until the first thread exits that atomic block. (Because we treat every instruction as being in an implicit single-instruction atomic block, we can think of the *serial* semantics as having a global re-entrant lock that each thread must acquire to enter any atomic block and that is released on exiting a block.)

Traditional parallel correctness properties such as *atomicity*, *serializability*, or *linearizability* hold for a program when, for any interleaved execution of the program, there exists a *similar* serial execution that produces an identical final program state. The differences between these correctness properties are in their definitions of *similar* executions:

- *Atomicity* [9] requires only that, for each interleaved execution, there exists a serial execution yielding the same final program state. The interleaved and serial executions need not be similar in any other way.
- *Serializability* [28] requires that, for each interleaved execution, there exists a serial execution which both yields the same final program state and in which all the same atomic blocks are executed.

  Conflict-serializability [28] further requires that all corresponding atomic blocks perform the same conflicting read and write operations in the interleaved and parallel execution, and that all pairs of conflicting operations occur in the same relative order. Conflict-serializability, though very strict, can be checked efficiently, and is thus used in many atomicity testing and verification tools, (e.g., [8–10, 12, 37]).
- *Linearizability* [17], like serializability, requires that, for each interleaved execution, there exists a serial execution which both yields the same final program state and in which the same atomic blocks are executed. Further, it requires that any pair of atomic blocks whose execution does not overlap in the interleaved execution must occur in the same order in the serial execution.

  Note that this definition of linearizability is somewhat different than that of [17] and later generalization [24], which formalize atomic blocks as having distinguished *responses* or *return values* and compare program states via observational equivalence—i.e. whether sequences of atomic blocks would return the same values. Our definition is appropriate for general atomic blocks without distinguished return values, while capturing the key requirement that a serial execution is equivalent only if it preserves the ordering of non-overlapping atomic blocks.

All three properties defined above require that, for each interleaved execution, there exists some serial execution producing an *identical* final state. As discussed in Section 2, such strict state equality is too restrictive to capture critical noninterference properties for many programs. For such programs, we propose employing a user-specified *bridge predicate* [3]—a predicate relating a pair of program states from an interleaved and a serial execution—to define a *semantic* equivalence between final program states.

That is, we can define *semantic atomicity*, *semantic serializability*, and *semantic linearizability*, all with respect to a user-specified bridge predicate Φ, by allowing in the above definitions that, for any interleaved execution with final state σ, the equivalent serial execution can have any final state σ' such that $\Phi(\sigma, \sigma')$.

The checking algorithm described in Section 4 tests semantic linearizability, primarily because linearizability is significantly easier to check, as it constrains the search space of similar serial executions to those that preserve the ordering of non-overlapping atomic blocks. But before describing out testing algorithm, we will briefly formalize in Section 3.2 these three parallel correctness properties.

3.2 Formal Definitions

In this section, we briefly formalize the above definitions of *semantic atomicity*, *serializability*, and *linearizability*.

Let *Thread* denote the set of program threads, Σ denote the set of program states including the thread-local states, *Atomic* denote the set of static program block *annotated* as atomic, and *Op* denote the set $\{\texttt{begin}(a) : a \in \mathit{Atomic}\} \cup \{\texttt{end}, \epsilon\}$ of atomic block operations. Intuitively, the operations are used to label the state transitions as follows. `begin(a)` marks the beginning of a dynamic instance of the static atomic block a, `end` marks the end of the last open atomic block, and ϵ is used for all other state transitions. We assume that atomic blocks are properly nested, and all instructions are inside one atomic block. If an instruction is not inside a programmer-annotated atomic block, then we assume that there is an implicit atomic block containing just that instruction.

Definition 1. *A program P consists of an initial program state σ_0 and a transition relation $\rightarrow$:*

$$\rightarrow \;\subseteq\; \Sigma \times \mathit{Thread} \times \mathit{Op} \times \Sigma$$

We write $\sigma \xrightarrow{t:\texttt{op}} \sigma'$ when the program can transition from state σ to σ' by executing the atomic block operation `op` by thread t.

Definition 2. *An* ***execution*** *of program $P = (\rightarrow, \sigma_0)$ is a sequence of transitions in the operational semantics $\rightarrow$:*

$$\sigma_0 \xrightarrow{t_1:op_1} \sigma_1 \xrightarrow{t_2:op_2} \cdots \xrightarrow{t_n:op_n} \sigma_n$$

An execution is ***complete*** *if all `begin(a)` have a matching `end` operation in the same thread.*

Definition 3. *An execution E is* ***serial*** *iff, for each matched t : $\texttt{begin}(a)$ and t : `end` transition in E, there are no transitions between the two by any other thread $s \neq t$.*

Definition 4. *A transition t : op in an execution E is a* ***top-level*** *transition if it does not occur between any matched t : $\texttt{begin}(a)$ and t : `end` by the same thread.*

Note that all top-level transitions are a `begin` or `end` since we assume that all instructions are part of atomic blocks.

Definition 5. *A complete execution $E = \sigma_0 \xrightarrow{\cdots} \sigma_n$ of program P is* ***semantically linearizable with respect to*** *Φ iff there exists a serial execution $E' = \sigma_0 \xrightarrow{\cdots} \sigma'_{n'}$ of P such that:*

(1) $\Phi(\sigma_n, \sigma'_{n'})$,

(2) for every thread $t \in$ Thread, the sequence of top-level operations performed by t is the same in E and E'.

(3) non-overlapping top-level atomic blocks in E appear in the same order E'.

A program P is ***semantically linearizable with respect to*** *Φ iff every execution of P is semantically linearizable.*

Note that if we remove conditions (2) and (3) above we obtain the notion of *semantic atomicity*. Similarly, if we remove only the condition (3) above we obtain the notion of *semantic serializability*. And when the bridge predicate $\Phi(\sigma, \sigma')$ is strict state equality $\sigma = \sigma'$, we obtain traditional linearizability, atomicity (removing conditions 2 and 3), and serializability (removing condition 3). We prefer to work with the stronger notion of linearizability because it is significantly easier to check.

Note that, for the purpose of checking similarity between the parallel and the serial executions, we identify the dynamic instances of atomic blocks by the combination of the thread that runs them, the static label of the atomic block, and the index of the dynamic occurrence in the execution.

4. Testing Semantic Linearizability

Now that we have defined semantic linearizability for programs with atomic block specifications, we can address the problem of checking the linearizability of such atomicity specifications.

Suppose P is a program with atomic blocks specified to be semantically linearizable with respect to bridge predicate Φ. Checking the linearizability of P consists of two problems:

(1) Given an interleaved execution E of a program P, is E semantically linearizable with respect to Φ?

(2) Is program P semantically linearizable with respect to bridge predicate Φ? That is, is every interleaved execution of P semantically linearizable?

Given a solution to the first problem, we can in theory solve the second by enumerating all interleaved executions of P and checking if each is linearizable. In practice, however, it is typically not feasible to enumerate all executions of a parallel program.

Instead, we resort to checking the linearizability of only a subset of the interleaved executions of P. Such a checking procedure will be *sound*—if we discover any executions of P that are not linearizable, then P cannot be linearizable—but *incomplete*—even if all checked executions are linearizable, we cannot know for certain that P itself is linearizable.

There are a number of existing techniques and tools that can be applied to generate a subset of the parallel, interleaved executions of a program for testing and verification—for example, a preemption-bounded model checker [25] such as CHESS [26] or an active testing [29, 30] tool such as CalFuzzer [18]. We describe in Section 5.1 the details of our technique for generating the interleaved executions to test.

The key to the effectiveness of our approach, however, is to consider only those interleaved executions in which only a small number of atomic blocks either have their execution *interrupted* by the operations of other threads or themselves interrupt the atomic blocks of other threads.

We show in the Sections 4.1 and 4.2 that we can efficiently check the linearizability of such *interruption-bounded* executions. And in Section 4.3 we will describe a technique, leveraging optional programmer-supplied hints to further increase the efficiency of such testing. Our experimental results demonstrate that testing a program for linearizability only on *interruption-bounded* interleaved executions is sufficient to find real atomicity errors.

4.1 Interruption-Bounded Executions

Let E be an interleaved execution of some program P:

$$E \;=\; \sigma_0 \xrightarrow{t_1:op_1} \sigma_1 \xrightarrow{t_2:op_2} \cdots \xrightarrow{t_n:op_n} \sigma_n$$

We say a top-level atomic block $t : \texttt{begin}(a_i), \ldots, t : \texttt{end}$ in thread t in E is *interrupted* if any operations by other threads occur between $t : \texttt{begin}(a_i)$ and $t : \texttt{end}$ in E. The interrupting operations in the other threads are part of atomic blocks that *interrupt* the atomic block $t : \texttt{begin}(a_i), \ldots, t : \texttt{end}$.

Suppose an execution E has R interrupt*ed* atomic blocks and K interrupt*ing* atomic blocks. (Note that a single block may be both interrupted and interrupting). We ask the question, how many possible linear orderings are there of the top-level atomic blocks of E that preserve the order of non-overlapping atomic blocks in E?

We show below in Theorem 6 that such an execution E has no more than $(K + 1)^R$ possible linear orderings of its top-level atomic blocks that preserve the order of non-overlapping atomic blocks. As we discuss in the next section, to check that an execution E is semantically linearizable, we will examine linear orderings of the top-level atomic blocks of E that preserve the ordering of non-overlapping blocks in E. Thus, if execution E is *interruption-*

bounded—i.e. has no more than R interrupted atomic blocks and no more than K interrupting blocks—then there will be no more than $(K+1)^R$ serial schedules that need to be examined.

Theorem 6. *Suppose an execution E has R top-level interrupt*ed *atomic blocks and K top-level interrupt*ing *atomic blocks. There are no more than $(K+1)^R$ possible linear orderings of the top-level atomic blocks of E that preserve the order of non-overlapping atomic blocks.*

Proof. The proof is by induction on R. For the base case $R = 1$, the bound is $K + 1$, because the K interrupting blocks are themselves non-overlapping and thus their linear order is fixed. The interrupted block can be placed in $K + 1$ positions in the order.

Suppose E has R interrupted atomic blocks and K interrupting atomic blocks. There exists some set S of $c \geq 1$ blocks in E such that: (1) every block in S is interrupted, and (2) no block in S interrupts any block not in S.

Suppose that such an S exists with $c = |S| = 1$. Then, there are no more than $(K+1)^{R-1}$ linear orderings of the remaining blocks, with the single block in S removed. And there are no more than $K + 1$ ways to add back the single block into any such order, yielding the desired bound.

Suppose instead that an S exists only with $c = |S| > 1$. Then every block in S is both interrupt*ed* and interrupt*ing*. (If any block were not interrupting, then it would be an S with $c = 1$.) Thus, there are no more than $(K + 1 - c)^{R-c}$ linear orderings of the remaining atomic blocks, of which $R - c$ are interrupted and no more than $K - c$ are interrupting. Consider the number of distinct ways in which the c blocks of S could appear in such an ordering of the remaining blocks. There are $c!$ linear orderings of the c blocks of S. And, relative to the remaining $K - c$ interrupting blocks, there are no more than $K - c + 1$ possible positions for each of the c blocks in S. Thus, the number of ways to add one linear ordering of the c blocks of S to one linear ordering of the remaining blocks is no more than $K!/c!(K-c)!$, which is the the number of ways to partition a sequence of length c into $K - c + 1$ segments, allowing empty segments. The desired bound holds, as:

$$(K+1-c)^{R-c} \cdot c! \cdot \frac{K!}{c!(K-c)!} \quad \leq \quad (K+1)^R \qquad \square$$

4.2 Testing Linearizability of Interruption-Bounded Executions

Algorithm 1 lists *CheckLinearizable*(P, Φ, E), our algorithm for testing the semantic linearizability, with respect to Φ, of an execution E of a program P.

We say that a *schedule* is a sequence $(t_1, a_1), \ldots, (t_n, a_n)$ of pairs of thread identifiers t_i and atomic block labels a_i. We assume that all sources of nondeterminism in a program P, besides the scheduling of parallel threads, have been eliminated. For example, the input to P and the environment in which P runs must be fixed. Thus, the behavior of the serial executions of P, in which no atomic block interrupts the execution of any other block, are uniquely identified by the *schedule* in which the top-level atomic blocks occur.

Then, let *Linearizations*(E) be a procedure computing the set of all schedules of the top-level atomic blocks in E that preserve the order of non-overlapping atomic blocks in E. A serial execution can be a witness to the linearizability of E only if it corresponds to one of the schedules in *Linearizations*(E). By Theorem 6, the number of such schedules, and thus the number of such serial executions, is bounded by the number of interruptions in E.

Algorithm 1 *CheckLinearizable*(P, Φ, E)

```
$\sigma \leftarrow$ final state of execution $E$
for $s \in$ Linearizations$(E)$ do
    if Execute$(P, s)$ succeeds, yielding $\sigma'$ then
        if $\Phi(\sigma, \sigma')$ then
            return true
        end if
    end if
end for
return false
```

We need only a mechanism for controlling the execution of a program P to force it along a schedule s. Let *Execute*(P, s) denote such a procedure. At a high level, for a program P and a schedule $s = (t_1, a_1), \ldots, (t_n, a_n)$, procedure *Execute*$(P, s)$ will, for each i from 1 to n:

- If thread t_i is not active or the next top-level atomic block to be started by t_i is not labeled a_i, then *Execute*(P, s) fails.
- Otherwise, we let thread t_i execute `begin`(a_i) and let it continue to run until it executes a matching `end`. If thread t_i blocks, *Execute*(P, s) fails.

  Similarly, *Execute*(P, s) fails if t_i runs forever without ever reaching a matching `end`. As the termination of t_i is undecidable, the best we can do is for *Execute*(P, s) to fail after t_i does not reach a matching `end` in a specified amount of time or number of instructions.

4.3 Hints for More Efficient Testing

In testing the semantic linearizability of the atomic blocks in a program P, we expect to have to test the linearizability of many interleaved executions of P. We expect that the great majority of these tested interleavings will be linearizable—concurrency errors such as atomicity violations tend to occur only on a small fraction of executions, especially in well-tested and widely-used software. (Our experimental results match this expectation.)

If an interleaved execution E is *not* linearizable, then we will have to look at all serial ways to schedule the top-level atomic blocks of E that are consistent with the ordering of non-overlapping blocks in E. But if an interleaved execution E *is* linearizable, we can determine this fact by finding a single equivalent serial execution. This raises the possibility that, for executions that turn out to be linearizable, we could make the testing procedure described in Section 4.2 more efficient by prioritizing the order in which we examine the possible linearizations of E.

Thus, we propose two kinds of optional hints that a programmer can add to their multithreaded code, along with their semantic atomicity specification. For an interleaved execution E, the hints will suggest which serial orderings of the overlapping atomic blocks should give equivalent results. Before falling back to a complete search of all linearizations of E, we will first try the linearizations consistent with these hints from the programmer.

Our optional hints take two forms: (1) linearization points, and (2) distinguished reads and writes:

Linearization Points First, a user can specify *linearization points* (also called *commit points*) for atomic blocks. Any dynamic execution of an atomic block should reach at most one annotated linearization point.

This hint indicates that, if two atomic blocks overlap and both execute a linearization point, then the block that executed its linearization point first should be ordered first in any serial execution.

Manually-annotated linearization points are often used in efforts to prove or verify the correctness of concurrent data structures [5, 7, 35, 38]. However, it has been observed [2, 5, 35] that it may be very difficult to identify or annotate all linearization points for some programs.

Distinguished Reads and Writes Second, a user can annotate certain reads and writes of shared variables as *distinguished* reads and writes. When linearization points cannot be identified statically, one could often identify some distinguished shared memory accesses (i.e. reads and writes) whose ordering determines the ordering between the atomic blocks. For example, if an atomic block inserts (i.e. writes) an item to a list and another overlapping atomic block gets (i.e. reads) the same item from the list, then the ordering between the write and read accesses determines the ordering between the atomic blocks. If a CAS operation succeeds, then a shared memory write performed by the CAS is considered distinguished. On the other hand, if a CAS operation fails, then the shared memory read performed by the CAS operation can be ignored (i.e. not considered distinguished). In several of our benchmarks, we have found that even if we cannot identify the linearization points of all atomic blocks, we can identify distinguished reads and writes and use them to determine the ordering among overlapping atomic blocks. We next describe how we use distinguished operations to order atomic blocks.

Given *distinguished* operations op_1 and op_2 on the same shared variable, we say that op_1 is *ordered before* op_2 if at least one of the two operations is a write and if op_1 is executed first.

Suppose atomic blocks B_1 and B_2 overlap. These hints indicate that B_1 should be ordered before B_2 in any serial execution if, for any variable v, some distinguished write to v in B_1 or the last distinguished read to v in B_1 is ordered before a distinguished write to v in B_2 or the last distinguished read of v in B_2.

These hints may indicate that B_1 should come before B_2 and that B_2 should come before B_1, in which case we ignore the distinguished reads/writes for ordering B_1 and B_2 with respect to each other.

Using Hints in Testing Linearizability Given an interleaved execution E, we use a depth-first search to find a serial ordering consistent with the annotated hints in E. And if no execution is consistent including both the *linearization points* and the *distinguished reads and writes*, we find an ordering consistent with just the *linearization points*. We test this single serial ordering to see if it is a witness to the semantic linearizability of E, and, if not, we fall back to the exhaustive search in Section 4.2.

Our experimental results demonstrate that these hints can improve our linearizability testing to the point where the first serial execution to be examined is found to satisfy the bridge predicate. Furthermore, using these optimizations is *sound* because the testing procedure falls back to searching all other possible serial linearizations when a programmer's hints do not guide us to a witness to linearizability.

5. Evaluation

In this section, we describe our efforts to experimentally evaluate our approach to specifying and checking semantic atomicity for multithreaded programs. Specifically, we seek to demonstrate that:

1. We can find real atomicity errors in multithreaded programs by testing the semantic linearizability of random interleaved executions with a small number of interrupted and overlapping atomic blocks.
2. In the common case where a tested interleaving *is* linearizable, we can soundly increase the efficiency of our testing using optional programmer annotations.

5.1 Implementation

In order to evaluate our claims, we implemented our approach for Java programs. Our implementation consists of several components: (1) an annotation and assertion library for specifying which blocks of code in a Java program are intended to be semantically atomic, as well as for specifying the bridge predicate with respect to which the blocks are intended to be atomic; (2) a component to generate random, interruption-bounded interleaved executions of a multithreaded test program; (3) a component to test the semantic linearizability of a given interleaved execution by generating and examining all serial executions that are linearizations of the interleaved execution.

```
class Atomic {

  static void open()

  static void close()

  static void assert(Object o, Predicate p)

  interface Predicate {
      boolean apply(Object a, Object b)
  }
}
```

Figure 7. Core atomicity specification API.

Atomicity Assertion Library Figure 7 shows the core API of our atomic assertion library. A programmer calls `Atomic.open()` and `Atomic.close()` to indicate the beginning and end of semantic atomic blocks in their code. Each call to `open` is uniquely identified by its location in the program source (accessible by, e.g., examining a call stack trace).

The bridge predicate giving the desired semantic equivalence between interleaved and serial executions is specified via `Atomic.assert`. In an interleaved execution, a sequence of calls to `Atomic.assert(obj,pred)` indicates that there must exist some serial execution—a linearization of the interleaved execution—in which each corresponding call `Atomic.assert(obj',pred)` is such that `pred.apply(obj,obj')` returns true.

That is, suppose the n^{th} call to `Atomic.assert(obj,pred)` in an interleaved execution records the serialized value of object `obj`. (We require that all objects passed to `Atomic.assert` implement the `Serializable` interface so that this recording is possible. Most common objects in the Java standard library can be serialized in this way.) Then, in a serial execution, while testing the linearizability of this interleaved execution, the n^{th} call to `Atomic.assert(obj',pred)` reads the previously serialized object `obj` and checks if `pred.apply(obj,obj')` holds. The serial execution is reported to be equivalent to the interleaved execution iff the same number of `Atomic.assert` calls are made and `pred.apply(obj,obj')` returns true for each one.

Sampling Interleaved Executions Our tool for randomly generating interleaved executions of a multithreaded test program is built on top of the publicly-available and open-source CalFuzzer [18] framework for testing concurrent Java programs. CalFuzzer uses Soot [33] to instrument Java bytecode, adding calls to a user's analysis/testing code on every read, write, lock, unlock, etc.—we use these calls to take control of the parallel scheduling of a Java program and replace it with our own scheduler.

Our thread scheduler is parameterized by a maximum number R of atomic blocks to *interrupt*, a number K of other atomic blocks to execute while the atomic block is interrupted, and a bound C on the number of times to interrupt at each distinct program statement. After any statement executes in the test program, the scheduler picks the next thread to execute a statement as follows:

Benchmark	Approx. LoC (Benchmark + Library)	# Static Atomic Blocks	Interruption-Bounded Interleavings			Avg. # of Serial Executions			Conflicts
			total	non-linear.	errors	linear.	linear. (heuristics)	non-linear.	
`ConcurrentLinkedQueue`	200	6	241	7	2	2.96	1.20	4.29	4
`ConcurrentSkipListMap`	1400	6	487	6	2^*	2.54	-	4.83	4
`ConcurrentSkipListSet`	100	6	463	5	2^*	2.57	-	4.6	4
`CopyOnWriteArrayList`	600	6	222	0	0	6.23	1.0	-	0
`CopyOnWriteArraySet`	60	6	221	0	0	4.39	1.0	-	0
`LockFreeList`	100	6	319	57	1	2.08	-	3.46	2
`LazyList`	100	8	231	0	0	2.46	1.02	-	2
`PJ pi`	150 + 15,000	1	20	5	1	1.0	-	4.8	1
`PJ keysearch`	200 + 15,000	1	904	0	0	1.0	-	-	0
`PJ mandelbrot`	250 + 15,000	1	73	0	0	1.0	-	-	0
`PJ phyl`	4400 + 15,000	2	605	27	1	1.0	-	125.56	2

Table 1. Experimental results.

- If the last statement was a top-level `Atomic.close` and no thread has an open atomic block, then pick the next thread randomly from among all active threads.
- If the last statement was by thread t and thread t has an open atomic block, then subject to certain constraints, we *interrupt* the atomic block thread t is executing, selecting a random different active thread to run next.

  The constraints are: (1) We perform no more than R interruptions during an execution. (2) For each statement in each possible calling context, we interrupt at that statement only if it is in the first C occurrences in the current execution of the statement and calling context, and only if we have not interrupted at that statement, calling context, and occurrence combination in any other run.
- If the last statement was a top-level `Atomic.close` and other threads have open, interrupted blocks:

  If we have executed K complete atomic blocks since interrupting the longest-open atomic block, we select, if possible, a random active thread *in an interrupted atomic block* to run next.

  Otherwise, randomly select to execute next any active thread *not* in an interrupted atomic block.

Overall, generated interleaved executions will have roughly K^R expected possible linearizations. In our experiments, we use parameters $R = 1$, $K = 4$, and $C = 4$.

Checking if an Interleaving is Linearizable Recall from the previous section that we can use CalFuzzer [18] to control the scheduling of a parallel Java application. We use this ability to implement procedure *Execute*(P, s), described in Section 4.2, for executing a program P along a serial schedule $s = (t_1, a_1), \ldots, (t_n, a_n)$.

We then implement Algorithm 1, given this `Execute`(P, s) procedure and using the atomic assertion library described above to check the specified bridge predicate.

5.2 Benchmarks

We evaluated our approach on a number of Java benchmarks. The name, size, and number of static blocks specified as atomic is given for each of these benchmarks in Table 1.

The first group of benchmarks are concurrent data structures from the Java standard library `java.util.concurrent` and elsewhere. `ConcurrentLinkedQueue`, `ConcurrentSkipListMap`, `ConcurrentSkipListSet`, and `CopyOnWriteArrayList` are from the Oracle Java SDK 6 (update 20). `LockFreeList` is a concurrent, lock-free list from [16], used as a benchmark by [4]. Benchmark `LazyList` is a concurrent set, implemented as a linked list with lazy deletion, from [35].

As our technique is designed to be applied to whole, closed programs, we must create a test harness for each data structure benchmark. Each harness creates one instance `obj` of the data structure and then calls four to eight methods on the instance in parallel, recording the return values. Each method call is specified to be semantically atomic with respect to a bridge predicate requiring both that `obj.equals(obj’)` and that all method return values be the same. For additional details, please see our benchmarks and test harnesses, which can be downloaded from: `http://www.cs.berkeley.edu/~jburnim`.

The other group of benchmarks are from the Parallel Java (PJ) Library [19]. The PJ benchmarks include an app for computing a Monte Carlo approximation of π (`pi`), a parallel cryptographic key cracking app (`keysearch3`), an app for parallel rendering of Mandelbrot Set images (`mandelbrot`), and a parallel branch-and-bound search for optimal phylogenetic trees (`phylogenetic`). Each of these benchmarks relies on roughly 15,000 lines of PJ library code for threading, synchronization, etc.

5.3 Experimental Setup and Results

For each benchmark, we execute our systematic random scheduler, described in Section 5.1 to generate a number of interleaved, *interruption-bounded* executions. We test each generated execution to see if it is semantically linearizable.

The number of *interruption-bounded* executions generated for each benchmark is listed in Column 4. Column 5 lists the number of executions found to *not* be semantically linearizable. We found non-linearizable executions for four of the data structure benchmarks and two of the PJ application benchmarks. In Column 6, we report the number of distinct bugs exposed by these atomicity-violating executions. We discuss some of these errors in detail in the next section. Note that every violation of the linearizability of our semantic atomic blocks indicated a true error.

For the linearizable executions of each benchmark, Columns 7 and 8 report the average number of serial executions that had to be examined to find a witness to the linearizability, without and with heuristically using any hints from programmer annotations[2]. Our annotations greatly reduce the number of serial executions that must be examined for several of the data structure benchmarks.

[2] Note that, for the PJ benchmarks, because the different atomic blocks are largely independent, it is usually the case that the first serial execution we examine witnesses the linearizability.

Column 9 reports the number of serial executions examined for non-linearizable interleavings. For most benchmarks, this number is small, as expected, because we are testing the linearizability of *interruption-bounded* executions. [3]

Finally, Column 10 reports the number of atomic blocks in each data structure that are not conflict-serializable. We discuss these numbers in the next section.

5.4 Atomicity Errors Found

We now discuss several atomicity errors found by our testing.

ConcurrentLinkedQueue Our automated testing of our semantic atomicity specification for Java's ConcurrentLinkedQueue found two errors. As far as we can determine, these errors have not previously been reported.

The code in Figure 8 gives a simple test harness that exposes one of the two errors. Initially, queue q contains elements 1 and 2. We expect that, because methods add, remove, and size should all be atomic, in any parallel, interleaved execution the call to q.size() must return that the queue contains one element (after q.remove(1) but before q.add(3)) or two elements (before the remove or after the add). However, it is possible for q.size() to incorrectly report that the queue q contains **three** elements!

This source of this error is that computing the number of elements in a ConcurrentLinkedQueue requires traversing its internal linked list structure and counting the number of elements. Suppose thread2's call to q.size() begins its traversal, finding and counting elements 1 and 2. But, before the call sees that it is at the tail of the list, it is interrupted by thread1. The calls by thread1 to q.remove(1) and then q.add(3) eliminate element 1 from the head of the list and insert element 3 at the tail of the list. Then, when thread2's call to q.size() continues, it finds and counts element 3 and returns that queue contains three items.

Our testing found a similar error for the toArray method of ConcurrentLinkedQueue. Further, while our test harness only exercised the add, remove, size, and toArray methods of ConcurrentLinkedQueue, manual inspection of its source code revealed that methods equals and writeObject can similarly return non-atomic results due to iterating over the elements of the queue without checking for concurrent modifications.

We note that although ConcurrentLinkedQueue's documentation[4] specifies that iteration through such a queue is only "weakly consistent", no such warning is given for methods size, toArray, equals, or writeObject. In fact, the documentation for the size method states:

> "Beware that, unlike in most collections, this method is NOT a constant-time operation. Because of the asynchronous nature of these queues, determining the current number of elements requires an O(n) traversal."

which seems to specify that size, although it requires $O(n)$ time, will return a consistent value (i.e., be linearizable). We thus judge that the unexpected behaviors of these methods are errors.

```
      Queue q = new ConcurrentLinkedQueue();
      q.add(1);  q.add(2);
      int sz = 0;

    thread1:                 thread2:
     @assert_atomic {
        q.remove(1);          @assert_atomic {
     }                           sz = q.size();
     @assert_atomic {         }
        q.add(3);
     }
            bridge predicate:
               q.equals(q') && (sz == sz')
```

Figure 8. Simple harness for ConcurrentLinkedQueue that reveals an atomicity error involving add, remove, and size. The annotated blocks are *not* semantically linearizable with respect to the bridge predicate.

ConcurrentSkipListMap and Set Our testing of benchmarks ConcurrentSkipListMap and ConcurrentSkipListSet found two violations of our semantic atomicity specifications for each benchmark. In particular, our test harnesses for these benchmarks each concurrently performs two insertions, two deletions, a call to size(), and a call to toArray() (or keySet().toArray() for ConcurrentSkipListMap). Our specification asserts that all six method calls execute semantically as if atomic, and our testing finds that neither method size nor method toArray is semantically atomic.

Note that the documentation[5] for ConcurrentSkipListMap and ConcurrentSkipListSet do warn for method size:

> "Additionally, it is possible for the size to change during execution of this method, in which case the returned result will be inaccurate. Thus, this method is typically not very useful in concurrent applications."

Thus, our specification is too strict in this case, as method size is not expected to be semantically atomic. Further, the documentation makes it clear that some bulk methods such as equals, putAll, etc., are not intended to be atomic. It is not clear from this documentation whether or not toArray is intended to be atomic.

Lock-Free List Our automated testing of our semantic atomicity specification for the lock-free list from [16] found one previously known error. In this lock-free list, two concurrent calls to remove can incorrectly both report that they have successfully deleted the same single element from a list. The online errata to [16] corrects this error.

PJ phyl Branch-and-Bound Search We also found a previously-unknown atomicity error in Parallel Java benchmark phyl. Figure 9 presents a very simplified, high-level version of the benchmark that illustrates the nature of the error.

Benchmark phyl is a parallel branch-and-bound search to find a minimum-cost phylogenetic tree for a given collection of DNA sequences. The search is nondeterministic—there may exist multiple minimum-cost trees and the search could return different minimum-cost trees on different runs, depending on the thread schedule and the resulting order in which the candidate trees are evaluated.

The benchmark can be thought of as parallel for-loop over possible phylogenetic trees. For each tree t, the cost is computed and the global minimum cost min_cost is updated. This update is safe, as proper synchronization is used to protect updates to the minimum cost. Thus, the final value of min_cost will always be correct.

Updates to min_tree, however, are not properly synchronized. Suppose one parallel loop iteration finds a new minimum-cost tree, updates min_cost, and enters the body of the if-statement with

[3] Difficulties controlling the Java scheduler benchmark phyl, however, lead to extra, unwanted interruptions and thus a larger number of serial executions that must be examined.

[4] http://download.oracle.com/javase/6/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html

[5] http://download.oracle.com/javase/6/docs/api/java/util/concurrent/ConcurrentSkipListMap.html and Set.html

```
parallel-for (t in trees) {
   @assert_atomic {
      cost = compute_cost(t)
      synchronized (min_cost) {
         min_cost = min(min_cost, cost)
      }
      if (cost == min_cost) {
         min_tree = t
      }
   }
}
      bridge predicate:
         min_tree.equals(min_tree')
            && (min_cost == min_cost')
```

Figure 9. Simplified version of PJ `phyl` benchmark highlighting the nature of the atomicity error found by our technique.

condition `cost == min_cost`, but is then interrupted by some other parallel loop iteration. The other iteration could further decrease `min_cost` and write to `min_tree`. But then when the first loop iteration continued, it would incorrectly overwrite `min_tree` with a tree no longer of minimal cost. Such an inconsistent value for `min_tree` cannot occur in a serial execution in which the body of iteration of the parallel `for`-loop occurs atomically.[6]

5.5 Discussion

Comparison to Conflict-Serializability Most existing tools for detecting atomicity violations check whether executions of a test program are conflict-serializable. As discussed in Section 3.1, conflict-serializability is a strict notion of atomicity, requiring that, for each interleaved execution, there exist a serial run in which every atomic block executes the same set of conflicting read and write operations, all in the same relative order.

Column 10 of Table 1 shows that several of the data structure benchmarks had atomic blocks that were not conflict-serializable, despite being semantically linearizable. Note that this column reports the number of *static* atomic blocks found to not be conflict-serializable in at least one dynamic, parallel execution. When run on all of the interruption-bounded interleavings in our experiments, a traditional atomicity analysis based on conflict-serializability would report 100+ dynamic atomicity violations for most of these benchmark. These would all be false positives (except for the few also violating semantic atomicity) that a user would have to examine. Burckhardt et al. [2] have also found that traditional atomicity analyses produce hundreds of false warnings for similar concurrent data structures.

Every atomicity violation reported by our approach, on the other hand, indicated a real atomicity violation, leading to program results not equivalent to those of any serial execution.

Effectiveness of Interruption-Bounded Testing Our experimental results demonstrate that we can find many real semantic atomicity errors by testing linearizability on interleaved executions with a small number of interruptions. There are some errors, however, that require more interruptions to detect. For example, there is a known[7] atomicity violation in the version of Java's `ConcurrentLinkedQueue` we tested (JDK 6, update 20), involving concurrent calls to `poll()` and `remove(o)`. In particular, it is possible for a call to `remove(o)` to return true—indicating that it removed object `o` from the queue—while a parallel call to `poll()` appears to remove and return the same object `o`. But this error can occur only if the call to `remove(o)` locates `o` at the head of the queue, then `poll()` interrupts and starts to (non-atomically) remove `o` from the queue, and then `remove(o)` interrupts `poll()` and finishes its remove.

Comparison to Determinism The error we detect in the Parallel Java (PJ) `phyl` benchmark was missed by our previous work [3], which attempted to verify the semantic deterministic behavior of this and other benchmarks. In [3], we checked a semantic deterministic specification for the benchmark like:

```
deterministic {
   // Phylogenetic branch-and-bound search.
   ...
} assert (min_cost == min_cost');
```

This specification asserts the following. For any pair of executions E and E' of this code pair from initial state σ_0 to final states σ and σ', it must be the case that `min_cost` in σ equals `min_cost` in σ'. That is, letting Φ_{det} denote bridge predicate `min_cost == min_cost`, it asserts:

$$\forall \sigma_0 \xrightarrow{E} \sigma.\ \forall \sigma_0 \xrightarrow{E'} \sigma'.\ \Phi_{\text{det}}(\sigma, \sigma')$$

To compare, let Φ_{atm} denote our bridge predicate `min_tree.equals(min_tree')` $\wedge$ `(min_cost == min_cost')` for semantic linearizability. Then, our semantic atomicity specification is that, for any execution E from σ_0 to σ, there exists one serial execution E' from σ_0 to σ' such that $\Phi_{\text{atm}}(\sigma, \sigma')$ and E' is a linearization of E. That is:

$$\forall \sigma_0 \xrightarrow{E} \sigma.\ \exists \sigma_0 \xrightarrow{E'} \sigma'.\ \Phi_{\text{atm}}(\sigma, \sigma') \ \wedge\ E' \text{ a linearization of } E$$

This difference between existential and universal quantification in the specification is the core difference between the complementary notions of atomicity and determinism. Many parallel programs are intended to be deterministic—that is, to always produce semantically equivalent output for the same input, no matter the thread schedule. Deterministic specifications can exactly capture this intended determinism. But many parallel programs employ algorithms that are inherently nondeterministic and which can correctly return different final results in different runs on the same input. For example, the branch-and-bound search PJ `phyl`, which can correctly return different minimum-cost trees depending on the nondeterministic scheduling of its threads.

For such programs, a semantic atomicity specification can be thought of as specifying the nondeterministic behavior that is acceptable or intended—the results of any serial execution of the program, in which the atomic blocks may execute in a nondeterministic order but their executions cannot be interleaved. At the same time, the specification asserts that no additional nondeterminism—from the nondeterministic interleaving of specified atomic blocks in a parallel execution—should appear in the final results of the program. That is, the result of any interleaved execution must be semantically equivalent to the result of some serial execution.

Future Work Our experimental results provide promising evidence both that it is feasible to write semantic atomicity specifications for parallel applications and data structures, and that we can effectively test such specifications by checking them on interruption-bounded executions. Our parallel application benchmarks are of somewhat limited size, however, and much work remains to validate our approach on a wider range of programs. In particular, we must investigate what challenges larger applications pose to writing semantic atomicity specifications and to the scalability of our testing technique.

[6] The error in the `phyl` source is that each worker thread's call to `globalResults.addAll(results)` is not atomic. This method updates the global list of minimum-cost trees with each worker thread's list of locally-minimum-cost trees.

[7] `http://bugs.sun.com/view_bug.do?bug_id=6785442`

6. Related Work

A large body of work has focused on verifying and testing atomicity in multithreaded programs, including static verification via type systems [9, 10], dynamic detection of atomicity violations [1, 6, 8, 12, 36, 37], model checking of atomicity [14], and active testing [18, 30] for atomicity violations. These generally have focused on verifying that such atomic blocks are *conflict serializable* [28].

Some research efforts have focused on verifying and testing notions of atomicity that are less strict than conflict serializability of atomic sections. For example, [7] uses model checking to verify linearizable of atomic sections, but requires all atomic sections to be annotated with a linearization point which specifies the linear order in which the atomic sections must be executed in the matching serial execution. Similarly, [11] generalizes type-based verification of conflict-serializability (via reduction [22]) to account for non-conflict-serializable but side-effect-free operations like a failing compare-and-swap in a busy-waiting loop.

Another such weaker notion is *atomic-set serializability* [34], which groups storage locations into *atomic sets* and requires, for each atomic set, all atomic blocks are conflict-serializable with respect to the locations in the set. Violations of atomic-set serializability are detected dynamically by [13] and found through active testing [18, 30] by [21].

A related area of research is verifying and testing linearizability[17] for concurrent objects. A concurrent object is linearizable if, for any client program which interacts with the object only through its methods and for which all such method calls are specified to be atomic, all executions of the client are linearizable. Several efforts have manually proved linearizability for certain highly-concurrent data structures [5, 31, 32]. Further, [38], [35], Line-Up [2], and CoLT [4] model check linearizability of concurrent objects.

At a high level, our approach to testing semantic linearizability of atomic blocks is similar to the approach of Line-Up [2]. Line-Up generates all serial executions of a test harness using a given concurrent object, and then uses preemption-bounded model checking [25] to enumerate interleaved executions and to test that each such execution is equivalent to one of the pre-generated serial ones. Our approach, however, works on larger programs for which it is neither feasible to pre-generate all serial executions or to perform exhaustive preemption-bounded model checking, even with a small preemption bound. Rather, we randomly sample *interruption-bounded* interleaved executions and, for each such execution, examine only the corresponding serial executions (i.e. those with the same ordering of non-overlapping atomic blocks). Further, our approach is applicable to any program with annotated atomic blocks, not just concurrent objects, and our approach checks *semantic* linearizability.

A large body of work on *transactional memory* has developed hardware and software techniques for *implementing* atomic blocks. While such work provides hardware support, libraries, or language constructs that guarantee that blocks of code intended to be atomic are, in fact, executed atomically. Our work focuses instead on testing that a program correctly implements its intended atomicity. The kind of of semantic atomicity we test is analogous to transactional memory work on open nesting [27], transactional boosting [15], and coarse-grained transactions [20]. These lines of work achieve greater concurrency in running transactions in parallel by ignoring conflicts at the low level of reads and writes and focusing on whether data structure operations abstractly/semantically commute or conflict. For example, two calls to the `add` method of a concurrent list may conflict at both the level of individual reads and writes and when the data structure is viewed as an abstract list. But such calls can be seen to commute if the list is instead viewed as an abstract multiset.

7. Conclusion

The traditional notions of atomicity and linearizability require each interleaved execution to correspond to a serial execution that produces an identical final state. Our experiments show that the traditional interpretation of these properties is often too strong. Instead, we propose to allow the programmer to specify a bridge predicate that expresses a more relaxed, application-dependent notion of equivalence between the allowable final states.

The resulting semantic linearizability property is not only widely applicable but also effectively testable. We described and demonstrated experimentally one possible testing strategy, based on the observation that most atomicity bugs can be reproduced in parallel executions with a small number of atomic block interruptions, executions for which the set of candidate linear schedules is also small. This set of candidates can be further reduced by using programmer-annotated commit points in atomic blocks, to the point where in the common case we find the desired serial schedule on the first try. In our experiments all instances when a serial schedule could not be found were atomicity violations.

Acknowledgments

We would like to thank Tayfun Elmas, Benjamin Lipshitz, our shepherd Margo Seltzer, and our anonymous reviewers for their valuable comments on this paper. This research supported in part by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227), by NSF Grants CNS-0720906, CCF-1018729, and CCF-1018730, and by a DoD NDSEG Graduate Fellowship. Additional support comes from Par Lab affiliates National Instruments, NEC, Nokia, NVIDIA, Samsung, and Sun Microsystems.

References

[1] R. Agarwal, A. Sasturkar, L. Wang, and S. D. Stoller. Optimized run-time race detection and atomicity checking using partial discovered types. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 233–242. ACM, 2005.

[2] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. Line-Up: A complete and automatic linearizability checker. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 330–340. ACM, 2010.

[3] J. Burnim and K. Sen. Asserting and checking determinism for multithreaded programs. In *Proceedings of the 17th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 3–12. ACM, 2009.

[4] P. Černý, A. Radhakrishna, D. Zufferey, S. Chaudhuri, and R. Alur. Model checking of linearizability of concurrent list implementations. In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV)*, pages 465–479. Springer, 2010.

[5] R. Colvin, L. Groves, V. Luchangco, and M. Moir. Formal verification of a lazy concurrent list-based set algorithm. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV)*, pages 475–488. Springer, 2006.

[6] A. Farzan and P. Madhusudan. Monitoring atomicity in concurrent programs. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV)*, pages 52–65. Springer, 2008.

[7] C. Flanagan. Verifying commit-atomicity using model-checking. In *Proceedings of the 11th International SPIN Workshop on Model Checking of Software*, pages 252–266. Springer, 2004.

[8] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–267. ACM, 2004.

[9] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming*

Language Design and Implementation (PLDI), pages 338–349. ACM, 2003.

[10] C. Flanagan and S. Qadeer. Types for atomicity. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI)*, pages 1–12. ACM, 2003.

[11] C. Flanagan, S. N. Freund, and S. Qadeer. Exploiting purity for atomicity. *IEEE Transactions on Software Engineering*, 31(4):275–291, Apr. 2005.

[12] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 293–303. ACM, 2008.

[13] C. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic detection of atomic-set-serializability violations. In *Proceedings of the 30th ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 231–240. ACM, 2008.

[14] J. Hatcliff, Robby, and M. B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In *Proceedings of the 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 175–190. Springer, 2004.

[15] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216. ACM, 2008.

[16] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, Inc., 2008.

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

[18] P. Joshi, M. Naik, C.-S. Park, and K. Sen. An extensible active testing framework for concurrent programs. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV)*, pages 675–681. Springer, 2009.

[19] A. Kaminsky. Parallel Java: A unified API for shared memory and cluster parallel programming in 100% Java. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, page 231. IEEE Computer Society, 2007.

[20] E. Koskinen, M. Parkinson, and M. Herlihy. Coarse-grained transactions. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 19–30. ACM, 2010.

[21] Z. Lai, S. C. Cheung, and W. K. Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 235–244. ACM, 2010.

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

[23] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th annual ACM Symposium on Principles of Distributed Computing (PDOC)*, pages 267–275. ACM, 1996.

[24] N. Mittal and V. K. Garg. Consistency conditions for multi-object distributed operations. In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS)*, pages 582–. IEEE Computer Society, 1998.

[25] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 2007 ACM SIGPLAN Symposium on Programming Language Design and Implementation (PLDI)*, pages 446–455. ACM, 2007.

[26] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 267–280. USENIX Association, 2008.

[27] Y. Ni, V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 68–78. ACM, 2007.

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

[29] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 135–145. ACM, 2008.

[30] K. Sen. Race directed random testing of concurrent programs. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 11–21. ACM, 2008.

[31] V. Vafeiadis. Shape-value abstraction for verifying linearizability. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 335–348. Springer, 2009.

[32] V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 129–136. ACM, 2006.

[33] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 13–. IBM Press, 1999.

[34] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 334–345. ACM, 2006.

[35] M. Vechev, E. Yahav, and G. Yorsh. Experience with model checking linearizability. In *Proceedings of the 16th International SPIN Workshop on Model Checking Software*, pages 261–278. Springer, 2009.

[36] L. Wang and S. D. Stoller. Run-time analysis for atomicity. In *Proceedings of the 3rd Workshop on Runtime Verification (RV)*, pages 191–209. Elsevier, 2003.

[37] L. Wang and S. D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 137–146. ACM, 2006.

[38] J. M. Wing and C. Gong. Testing and verifying concurrent objects. *Journal of Parallel and Distributed Computing*, 17(1-2):164–182, Jan. 1993.