SideTrack: Generalizing Dynamic Atomicity Analysis

Jaeheon Yi
jaeheon@cs.ucsc.edu

Caitlin Sadowski
supertri@cs.ucsc.edu

Cormac Flanagan
cormac@cs.ucsc.edu

Computer Science Department
University of California at Santa Cruz
Santa Cruz, CA 95064

ABSTRACT

Atomicity is a key correctness specification for multithreaded programs. Prior dynamic atomicity analyses include *precise* tools, which report an error if and only if the observed trace is not serializable; and *imprecise* tools, which generalize from the observed trace to report errors that might occur on other traces, but which may also report false alarms.

This paper presents SideTrack, a lightweight online dynamic analysis that generalizes from the observed trace without introducing the potential for false alarms. If SideTrack reports an error, then some feasible trace of the source program is not serializable. Experimental results show that this generalization ability increases the number of atomicity violations detected by SideTrack by 40%.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification – *reliability*; D.2.5 [**Software Engineering**]: Testing and Debugging – *testing tools*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs – *specification techniques*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages – *program analysis*

General Terms

Languages, Algorithms, Verification

Keywords

Atomicity, serializability, dynamic analysis

1. INTRODUCTION

Recent trends in microprocessor architectures have made concurrency a central strategy for scaling up the performance of software. Unfortunately, writing correct multithreaded programs has proven to be very difficult because reasoning about program correctness must be done at the level of thread interleavings and platform-specific memory models. As our society depends increasingly on large software systems, reliability of multithreaded programs is an increasingly important concern. A crucial issue to consider is what it means for a multithreaded program to be correct, and how to specify correctness.

A general specification for multithreaded programs is *atomicity*, a strong non-interference property that is widely applicable amongst many different programs. Informally, a block of code is atomic if, for all executions of that code block, the effect of that execution can be considered in isolation from the rest of the running program. In the database literature this property is often called *serializability* [3].

Atomicity corresponds to a natural way of thinking about concurrency, where the problem of reasoning about interleavings between individual operations is reduced to reasoning about interleavings between much coarser atomic blocks. Inside an atomic block, simpler sequential reasoning may be applied to show correctness.

While atomicity is related to the notion of race-freedom [23], these two correctness properties provide complementary guarantees. Race-freedom guarantees that the program behaves as if executed on a sequentially-consistent memory model [1], while atomicity guarantees that each atomic block behaves as if executed serially.

In this paper, we focus on dynamically detecting atomicity violations in traditional multithreaded programs based on synchronization idioms such as locks, fork, join etc. Prior work on dynamic atomicity analysis includes both *precise* and *imprecise* tools.

- Precise dynamic atomicity analyzers, such as Velodrome and others [17, 11], never produce false alarms. Instead, they report an atomicity violation if and only if the observed trace is not serializable.

- In contrast, imprecise tools, such as Atomizer and others [13, 31, 34, 33], generalize from the observed trace to also report errors which may (or may not) occur on other possible traces. In practice, these imprecise tools may report false alarms, which are notoriously difficult to distinguish from real error reports.

Sound trace generalization.

This paper explores a third approach, which is to generalize from the observed trace without introducing the potential of false alarms. Even if the observed trace is serializable, our online analysis can still infer that the original program can generate other feasible traces that are not serializable.

PADTAD '09, July 19-20, 2009, Chicago, Illinois, USA.
Copyright 2009 ACM 978-1-60558-655-7/09/07 ...$10.00.

To illustrate this idea, consider the trace in Figure 1, where Thread 1 contains an atomic block with two synchronized statements, Thread 2 contains a single synchronized statement, and where the vertical ordering of the statements of the two threads reflects their relative execution order.

```
Thread 1          Thread 2
------------------------------

atomic {
  sync(m) {...}

  sync(m) {...}
}
                  sync(m) {...}
```

Figure 1: Observed Serial Trace

Clearly, this trace is serial and hence trivially serializable, and so a precise checker such as VELODROME would not detect any errors. A careful analysis of this trace, however, shows that the synchronized statement of Thread 2 *could have been scheduled* in between the two synchronized statements of Thread 1; the original source program that generated the trace of Figure 1 is also capable of generating the non-serializable trace shown in Figure 2.

```
Thread 1          Thread 2
------------------------------

atomic {
  sync(m) {...}
                  sync(m) {...}
  sync(m) {...}
}
```

Figure 2: Feasible Non-Serializable Trace

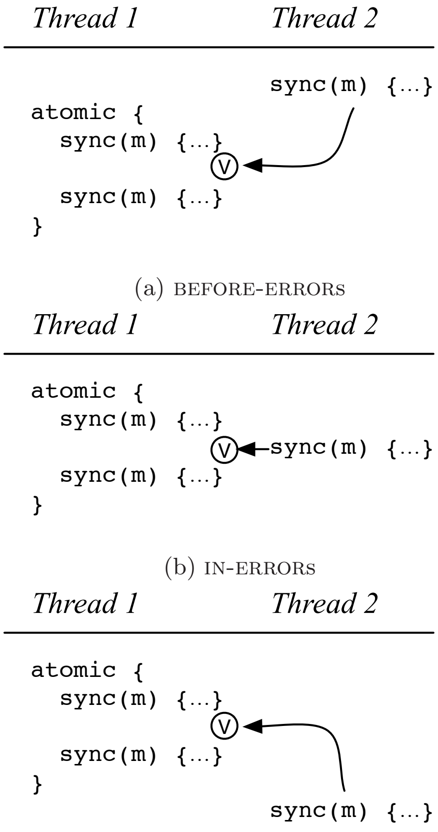
Technically, the synchronized block of Thread 2 could diverge in this alternate trace. Our analysis assumes that all synchronized blocks terminate, and that the program is free of race conditions. This latter assumption can be discharged by concurrently running an efficient precise race detector such as FASTTRACK [14] or Goldilocks [9].

We would like to catch the violation of Figure 2 without having to observe it directly. Whenever the same lock is acquired twice within a transaction, there is a *vulnerable window* between the two acquires where a *culprit* acquire by another thread could cause an atomicity violation.

SideTrack.

In this paper we introduce SIDETRACK, an online predictive atomicity analysis. By observing the serial trace of Figure 1, our dynamic analysis can detect that the source program contains an atomicity violation, even though that violation does not manifest itself in the current trace. Thus, our analysis generalizes from the observed trace to detect errors that *are guaranteed* to occur on another feasible trace. Furthermore, unlike prior imprecise tools, it performs this generalization without introducing false alarms.

Our analysis detects three kinds of errors – BEFORE-ERRORS, IN-ERRORS, and AFTER-ERRORS – depending on where the culprit acquire occurs in relation to the vulnerable window, as shown in Figure 3 (a), (b), and (c), respectively. Prior precise tools such as VELODROME only detect IN-ERRORS; SIDETRACK introduces the additional ability to detect BEFORE-ERRORS and AFTER-ERRORS[1].



(a) BEFORE-ERRORS

(b) IN-ERRORS

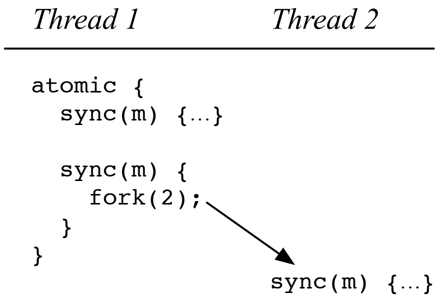
(c) AFTER-ERRORS

Figure 3: Three Kinds of Atomicity Violations

Although the above examples are rather straightforward, in practice generalizing from the observed trace in a sound manner may be rather involved. To illustrate some of the issues, consider an alternate trace shown in Figure 4, where the second synchronized block of Thread 1 forks Thread 2. In this situation, the synchronized statement of Thread 2 cannot be scheduled between the two synchronized statements of Thread 1.

Despite the similarities between Figures 1 and 4, the first trace reflects an atomicity error in the source program, whereas the second trace does not. SIDETRACK needs to perform a careful analysis of the happens-before relation of the observed trace to detect situations where certain synchronized blocks could have been scheduled before other synchronized blocks. SIDETRACK uses the standard technique of vector clocks to provide a compact and efficient representation of this happens-before relation. SIDETRACK tracks the relative timing of synchronization operations and flags an error if an operation by another thread is concurrent with one of the operations flanking a vulnerable window.

[1] Some tools do generalize to other traces using a combination of online and offline analysis; please see the related work section for more details.



```
Thread 1              Thread 2

atomic {
  sync(m) {...}

  sync(m) {
    fork(2);
  }
}
                      sync(m) {...}
```

Figure 4: Observed Serial Trace with Fork

Empirical validation.

On a range of standard multithreaded benchmarks, we show that generalizing from the observed trace significantly increases SideTrack's ability to detect atomicity violations, without introducing false alarms. In particular, SideTrack detected 27 in-errors, which would also be detected by prior precise tools such as Velodrome. However, SideTrack also predicted an additional 11 atomicity violations that did not occur on the observed trace, but which could occur on other feasible traces. Thus, the ability to soundly generalize from the observed trace increased the number of atomicity violations detected by SideTrack by 40%.

Summary.

This paper contains the following contributions:

- We show how serializable traces can still reveal atomicity violations in the original program.
- We present SideTrack, a lightweight on-line dynamic analysis tool that generalizes from the observed trace to predict atomicity violations that could occur on other feasible traces. SideTrack performs this generalization without introducing false alarms.
- We present experimental results showing that this predictive ability increases the number of atomicity violations detected by SideTrack by 40%.
- Our results also show that, despite performing on-line generalization, SideTrack offers performance competitive with other dynamic analyses.

2. MULTITHREADED PROGRAM TRACES

We begin by formalizing the notion of multithreaded program traces, which serve as the input to our dynamic analysis. A program consists of a number of concurrently executing threads that manipulate variables $x \in Var$ and locks $m \in Lock$. Each thread has a thread identifier $t \in Tid$. A *trace* α captures an execution of a multithreaded program by listing the sequence of operations performed by the various threads. The set of operations that a thread t can perform include:

- $rd(t, x)$ and $wr(t, x)$, which read and write from variable x;
- $acq(t, m)$ and $rel(t, m)$, which acquire and release a lock m;
- $begin(t, l)$ and $end(t, l)$, which demarcate an `atomic` block labelled l;
- $fork(t, u)$, which forks a new thread u; and
- $join(t, u)$, which blocks until thread u terminates.

We are concerned with the traces produced from an actual run of a source program. These traces will be *well-formed*; they fulfill expected constraints when forking, joining, acquiring and releasing. For example, every release operation has a corresponding acquire operation by the same thread earlier in the trace, and no operations by thread u occur in a trace prior to the forking of thread u.

Two operations in a trace *conflict* if they satisfy one of the following conditions:

- **Lock conflict:** they acquire or release the same lock.
- **Fork-join conflict**: one operation is either $fork(t, u)$ or $join(t, u)$ and the other operation is by thread u.
- **Program order conflict**: they are performed by the same thread.

The *happens-before relation* $<_\alpha$ on a trace α is the smallest transitively-closed relation on operations in α such that if operation a occurs before b in α and a conflicts with b, then a *happens-before* b and b *happens-after* a.

Two traces are *equivalent* if one can be obtained from the other by repeatedly swapping adjacent non-conflicting operations. Equivalent traces yield the same happens-before relation, and exhibit equivalent behavior.

A *transaction* in a trace α is the sequence of operations executed by a thread t starting with a $begin(t, l)$ operation and containing all t operations up to and including a matching $end(t, l)$ operation. To simplify some aspects of the formal presentation, we assume $begin(t, l)$ and $end(t, l)$ operations are appropriately matched and are not nested (although our implementation does support nested atomic specifications).

In a *serial* trace, all operations from each transaction are grouped together and not interleaved with the operations of any other transaction. A trace is serializable if it is equivalent to a serial trace.

If two operations in a trace have a fork-join or program order conflict, then we say they have an *enables conflict*. The *enables relation* $\prec_\alpha$ is the smallest transitively-closed relation on operations in α such that if operation a occurs before b in α and a has an enables conflict with b, then a *enables* b.

A lock operation a in α is *concurrent* with a later lock operation b by a different thread unless there exists an operation c between a and b such that $a <_\alpha c$ and $c \prec_\alpha b$. This is a tricky definition and is important for the later discussion, so it is worth developing the intuition behind it. In essence, in a hypothetical alternate trace where b occurs before a, c would also need to occur before a. However, once the happens-before edge between c and a is reversed, the thread which executed c may take an alternate path of execution. Since c enables b, this means that b may not occur if c occurs earlier.

In any good definition of concurrent, two concurrent operations may execute in either order. Therefore, to identify concurrent operations we need to ensure that a dependency does not exist between them. The happens-before relation identifies dependencies but is too restrictive; every pair of lock operations on the same lock are related by happens-before. The enables relation is too weak; the lock operations on m by different threads in Figure 5(a) are not related by enables, but it may not be possible to execute them in a different order (Figure 5(b)).

Vector clocks.

SideTrack uses vector clocks [22] to identify concurrent acquire operations within a trace. Vector clocks compactly represent the happens-before relation. A vector clock maps thread identifiers to integer clocks:

$$VC \stackrel{\text{def}}{=} Tid \rightarrow Nat$$

If vc is the vector clock for an acquire operation in a trace, then $vc(t)$ identifies the operations of thread t that happen-before that acquire.

Vector clocks are partially-ordered ($\sqsubseteq$) in a point-wise manner, with an associated join operation ($\sqcup$) and minimal element (c_0). In addition, the helper function inc_t increments the t-component of a vector clock:

$$\begin{array}{rcl}
vc_1 \sqsubseteq vc_2 & \text{iff} & \forall t.\ vc_1(t) \leq vc_2(t) \\
vc_1 \sqcup vc_2 & = & \lambda t.\ max(vc_1(t), vc_2(t)) \\
c_0 & = & \lambda t.\ 0 \\
inc_t(vc) & = & \lambda u.\ \texttt{if}\ u = t\ \texttt{then}\ vc(u) + 1\ \texttt{else}\ vc(u)
\end{array}$$

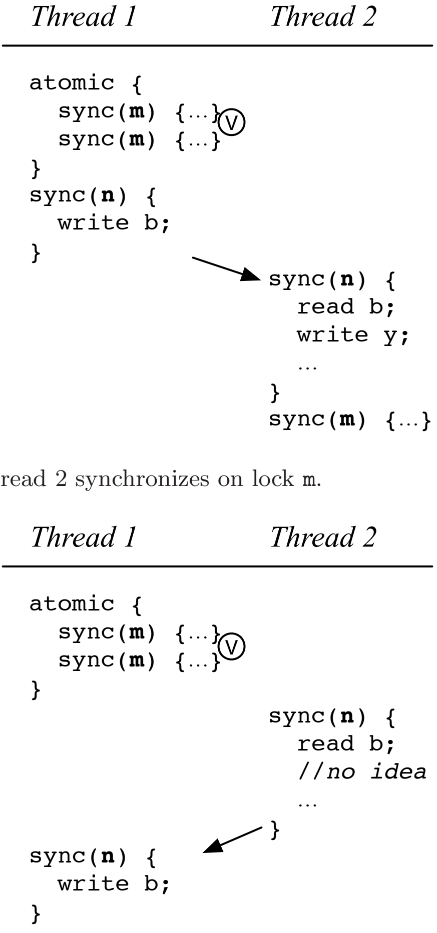
3. ANALYSIS

We refer to the actual trace perceived by our analysis tool as the *observed* trace. In addition to the observed trace, many other *feasible* traces exist for a given program. It is not generally possible to predict all feasible traces for a program without additional static information; for example, the observed trace may not have complete code coverage for the source program. Nevertheless, our analysis detects situations where a feasible but not observed atomicity violation can be predicted dynamically.

Whenever the same lock is acquired twice within a transaction, there is a *vulnerable window* between the two acquires where an acquire by another thread could cause an atomicity violation. Our analysis keeps track of these vulnerable windows, and flags an error if it detects that a feasible trace exists which exploits a vulnerable window to cause an atomicity violation. We refer to the external acquire which could occur in the vulnerable window as a culprit acquire.

Note that when predicting feasible traces dynamically, it is not possible to predict the entire trace. For example, we can predict that a lock acquire could have occurred between the two previous acquires, but the trace after that point may not be predictable based on the information obtained from the first trace. Instead, our analysis can only predict that *there exists* some alternate trace with an atomicity violation, where the alternate trace shares a prefix with the observed trace and executes certain synchronized blocks in a certain order.

Because of these inherent limitations in our ability to generalize from the observed trace, some traces which appear at first glance to have a predictable atomicity violation turn out to be more complicated upon closer inspection. For example, consider the trace in Figure 5(a), where it appears the synchronized block of Thread 2 could execute inside the vulnerable window of Thread 1. However, if the read of b by Thread 2 executes before the write to b by Thread 1, we can no longer predict the subsequent execution of Thread 2, and in particular cannot guarantee that Thread 2 will still synchronize on the lock m (Figure 5(b)). The notion of concurrency defined in Section 2 captures these constraints.



```
atomic {
  sync(m) {...}
  sync(m) {...}
}
sync(n) {
  write b;
}
                sync(n) {
                  read b;
                  write y;
                  ...
                }
                sync(m) {...}
```

(a) Thread 2 synchronizes on lock m.

```
atomic {
  sync(m) {...}
  sync(m) {...}
}
                sync(n) {
                  read b;
                  //no idea
                  ...
                }
sync(n) {
  write b;
}
```

(b) Thread 2 may not synchronize on lock m, due to an alternate control flow.

Figure 5: A non-vulnerable atomic block due to the happens-before edge on the lock n, which protects the variable b.

As stated in the introduction, our analysis detects three different kinds of errors (before-errors, after-errors, and in-errors), depending on where the culprit acquire occurs in relation to the vulnerable window.

In a before-error, the culprit acquire occurred before the vulnerable window, as illustrated in Figure 3(a). To detect before-errors, the analysis records the vector clock of the most recent acquire of each lock. When a lock is first acquired in an atomic block, the analysis checks if that acquire is concurrent with the previous acquire of that lock, via a vector clock comparison. If the two acquires are concurrent, then that lock is recorded as being *potentially interfering* in that atomic block; if the atomic block subsequently re-acquires that lock, then a before-error atomicity violation is reported.

Note that it is not enough to keep track of the most recent *release* of each lock. It is possible for a lock release by Thread 1 to be concurrent with an acquire of the same lock by Thread 2, while the corresponding acquire by Thread 1 enables the acquire by Thread 2. In the most extreme case, Thread 1 could fork Thread 2 while holding the the lock.

In an IN-ERROR, the culprit acquire occurs in the vulnerable window, as in Figure 3(b). Other dynamic atomicity analysis tools that do not generalize to additional traces typically catch IN-ERRORs; an IN-ERROR represents an actual atomicity violation in the observed trace. We catch IN-ERRORs inside the vulnerable window, when the program is about to execute the second acquire within the transaction. There is an atomicity violation if a thread is about to acquire the same lock twice within a transaction and discovers the last release of that lock is concurrent with the acquire about to happen[2].

In an AFTER-ERROR, the culprit acquire occurs after the vulnerable window, as in Figure 3(c). We catch AFTER-ERRORs when the program is about to execute the culprit acquire. We keep track of the most recent vulnerable window for every lock. If the vector clock for an acquire by a different thread is not later than the vulnerable window, than that thread could have executed the acquire in the vulnerable window.

Analysis formalization.

Based on these ideas, we now formally define our atomicity analysis as an online algorithm based on an analysis state $\sigma = (C, V, A, R, H, I)$ where:

- $C : \mathit{Tid} \rightarrow \mathit{VC}$ records the vector clock of the current operation by each thread;
- $V : \mathit{Lock} \rightarrow \mathit{VC}$ records the vector clock of the most recent vulnerable window for each lock;
- $A : \mathit{Lock} \rightarrow \mathit{VC}$ records the vector clock of the last acquire of each lock;
- $R : \mathit{Lock} \rightarrow \mathit{VC}$ records the vector clock of the last release of each lock;
- $H : \mathit{Tid} \rightarrow 2^{\texttt{LOCKS}} \cup \{\texttt{NotInX}\}$ records the set of locks held within the current transaction by each thread, or `NotInX` if that thread is not currently within a transaction; and
- $I : \mathit{Tid} \rightarrow 2^{\texttt{LOCKS}}$ records the set of potentially interfering locks for each thread.

In the initial analysis state, all vector clocks are initialized to c_0, except each C_t starts at $inc_t(c_0)$ to reflect that the first steps by different threads are not ordered.

$$\begin{array}{rcl} \sigma_0 & = & (\lambda t.\, inc_t(c_0), \\ & & \lambda m.\, c_0, \\ & & \lambda m.\, c_0, \\ & & \lambda m.\, c_0, \\ & & \lambda t.\, \texttt{NotInX}, \\ & & \lambda t.\, \emptyset) \end{array}$$

Figure 6 shows how the analysis state is updated for each operation a of the target program.

The first rule [ENTER] for $begin(t, l)$ records that thread t is in a new transaction by switching H_t away from `NotInX` to $\emptyset$, and resets the set of interfering locks. The complementary rule for $end(t, l)$ records that t is no longer in a transaction. Here, H is a function, H_t abbreviates the function application $H(t)$, and $H[t := V]$ denotes the function that is identical to H except that it maps t to V. Changes to the instrumentation state are expressed as functional updates for clarity in the analysis rules, but are implemented as in-place updates in our implementation.

Read and write operations do not affect the analysis state. If required, race conditions can be detected by running SIDETRACK concurrently with a race detector such as FASTTRACK [14]. We do not yet consider reads and writes of volatile variables which create nonblocking synchronization, although we believe we can adapt our analysis to handle these constructs.

We update vector clocks for fork and join operations to reflect the structure of the enables relation. The rule [FORK] for $fork(t, u)$ performs one "clock tick" for thread t and sets the vector clock associated with u to be greater than previous operations by thread t. The rule [JOIN] records that the last operation of the joined thread enables the join operation. The vector clock for the joined thread is incremented to preserve the invariant that the current vector clocks for each running thread are incomparable.

There are three analysis rules associated with an $acq(t, m)$ operation. All three rules share three common antecedents which:

1. update the vector clock for t to reflect that the current time for t is later than the previous release of m;
$$C' \quad = \quad C[t := C_t \sqcup R_m]$$
2. update the last acquire for m appropriately;
$$A' \quad = \quad A[m := C_t]$$
3. and report an AFTER-ERROR if the vulnerable window for m is not before the current clock for t:
$$\texttt{if}\ V_m \quad \not\sqsubseteq \quad C_t\ \texttt{then}\ \text{AFTER-ERROR}$$

The rule [OUTSIDE ACQUIRE] applies to acquires that are not within a transaction, and simply performs the above three actions.

The rule [FIRST ACQUIRE] applies the first time a thread acquires a lock within a transaction. We add m to the set of locks H_t acquired within that transaction, and check if the previous acquire of m was concurrent. If so, we add m to the set I_t of potentially interfering locks for that thread.

The rule [SECOND ACQUIRE] applies the second time a thread acquires a lock within a transaction. We record the new vulnerable window for m in V_m. We flag a BEFORE-ERROR if there is a previous interfering lock which could have executed after the first acquire by t. We flag an IN-ERROR if the previous release of m is not before the current clock for t. All previous operations by t, including the first acquire of m by t, are before the current clock of t. Therefore, if $R_m \not\sqsubseteq C_t$, then another thread released (and previously acquired) m between the two acquires by t.

The rule [RELEASE] for $rel(t, m)$ updates the vector clock associated with the latest release for m. We also perform one clock tick for thread t.

[2] Concurrent operations are by different threads, so this would mean that another thread acquired the lock.

Figure 6: SideTrack **Atomicity Violation Detection Algorithm**

[ENTER]

$$\frac{\begin{array}{rcl} H' & = & H[t := \emptyset] \\ I' & = & I[t := \emptyset] \end{array}}{(C,V,A,R,H,I) \Rightarrow^{begin(t,l)} (C,V,A,R,H',I')}$$

[EXIT]

$$\frac{H' \;=\; H[t := \texttt{NotInX}]}{(C,V,A,R,H,I) \Rightarrow^{end(t,l)} (C,V,A,R,H',I)}$$

[READ]

$$\frac{}{(C,V,A,R,H,I) \Rightarrow^{rd(t,x,v)} (C,V,A,R,H,I)}$$

[WRITE]

$$\frac{}{(C,V,A,R,H,I) \Rightarrow^{wr(t,x,v)} (C,V,A,R,H,I)}$$

[FORK]

$$\frac{C' \;=\; C[t := inc_t(C_t), u := C_t \sqcup C_u]}{(C,V,A,R,H,I) \Rightarrow^{fork(t,u)} (C',V,A,R,H,I)}$$

[JOIN]

$$\frac{C' \;=\; C[t := C_t \sqcup C_u, u := inc_u(C_u)]}{(C,V,A,R,H,I) \Rightarrow^{join(t,u)} (C',V,A,R,H,I)}$$

[FIRST ACQUIRE]

$$\frac{\begin{array}{rcl} H_t & \neq & \texttt{NotInX} \qquad m \notin H_t \\ \texttt{if } V_m & \not\sqsubseteq & C_t \texttt{ then } \text{AFTER-ERROR} \\ C' & = & C[t := C_t \sqcup R_m] \\ A' & = & A[m := C_t] \\ H' & = & H[t := H_t \cup \{m\}] \\ I' & = & (\text{if } A_m \sqsubseteq C_t \text{ then } I \text{ else } I[t := I_t \cup \{m\}]) \end{array}}{(C,V,A,R,H,I) \Rightarrow^{acq(t,m)} (C',V,A',R,H',I')}$$

[SECOND ACQUIRE]

$$\frac{\begin{array}{rcl} m & \in & H_t \\ \texttt{if } m & \in & I_t \texttt{ then } \text{BEFORE-ERROR} \\ \texttt{if } R_m & \not\sqsubseteq & C_t \texttt{ then } \text{IN-ERROR} \\ \texttt{if } V_m & \not\sqsubseteq & C_t \texttt{ then } \text{AFTER-ERROR} \\ C' & = & C[t := C_t \sqcup R_m] \\ V' & = & V[m := V_m \sqcup C_t] \\ A' & = & A[m := C_t] \end{array}}{(C,V,A,R,H,I) \Rightarrow^{acq(t,m)} (C',V',A',R,H,I)}$$

[OUTSIDE ACQUIRE]

$$\frac{\begin{array}{rcl} H_t & = & \texttt{NotInX} \\ \texttt{if } V_m & \not\sqsubseteq & C_t \texttt{ then } \text{AFTER-ERROR} \\ C' & = & C[t := C_t \sqcup R_m] \\ A' & = & A[m := C_t] \end{array}}{(C,V,A,R,H,I) \Rightarrow^{acq(t,m)} (C',V,A',R,H,I)}$$

[RELEASE]

$$\frac{\begin{array}{rcl} C' & = & C[t := inc_t(C_t)] \\ R' & = & R[m := C_t] \end{array}}{(C,V,A,R,H,I) \Rightarrow^{rel(t,m)} (C',V,A,R',H,I)}$$

Examples.

Figure 7 shows how vector clocks are updated on a sample trace fragment involving an AFTER-ERROR. The I and A parts of the analysis state play no part in discovery of AFTER-ERRORs, so they are omitted from the figure for simplicity. After the first release by thread 0, C_0 is incremented from <5,0. . . > to <6,0,. . . > and R_m is set to <5,0. . . >. When m is acquired a second time by thread 0, V_m is set to the current time for thread 0 (<6,0,. . . >). At the second release by thread 0, C_0 is incremented again from <6,0. . . > to <7,0,. . . > and R_m is updated to <6,0. . . >. When thread 1 goes to acquire m, an AFTER-ERROR is reported because $V_m \not\sqsubseteq C_1$. Once the AFTER-ERROR is reported, C_1 is joined with the time of the last release (R_m).

The example in Figure 8 illustrates that AFTER-ERRORs are more amenable to detection than BEFORE-ERRORs. SideTrack finds the AFTER-ERROR in the bottom trace because the acquires of `m` by the two threads are concurrent. In the top trace, the synchronization on `n` means that the acquires of `m` are *not* concurrent: the acquire of n by thread 2 (and, by transitivity, the acquire of m by thread 2) happens-before the acquire of n by thread 1 and the acquire of n by thread 1 enables the acquires of m by thread 1. Therefore, a BEFORE-ERROR is not detected in this case.

4. IMPLEMENTATION

We have developed a prototype implementation, called SideTrack, of our dynamic atomicity analysis. SideTrack is a component of RoadRunner, a framework designed for developing dynamic analyses for multithreaded programs. RoadRunner supports chaining of tools, so that multiple analyses can be run concurrently. RoadRunner instruments the target bytecode of a program during load time. The instrumentation code generates a stream of events for lock acquires and releases, field and array accesses, method entries and exits, etc[3]. RoadRunner tools, such as SideTrack, process this event stream as it is generated. RoadRunner enables analysis tools to attach instrumentation state to each thread, lock object, and data memory location used by the target program. Tool-specific event handlers update the instrumentation state for each operation in the observed trace and report errors when appropriate.

Analysis implementation.

By default, SideTrack checks for the specification that all methods are atomic. This lifts transactions to the method call level, and nested transactions represent nested method calls.

Each of the analysis rules in Figure 6 consists of one or more antecedents and a single consequent. Three rules are necessary to describe the event handler for acquires. When implementing these rules, it is useful to break down the antecedents into *conditionals*, *checks*, and *updates*. Conditionals indicate where the code must branch in an event handler.

[3] Re-entrant lock acquires and releases are redundant and are filtered out by RoadRunner.

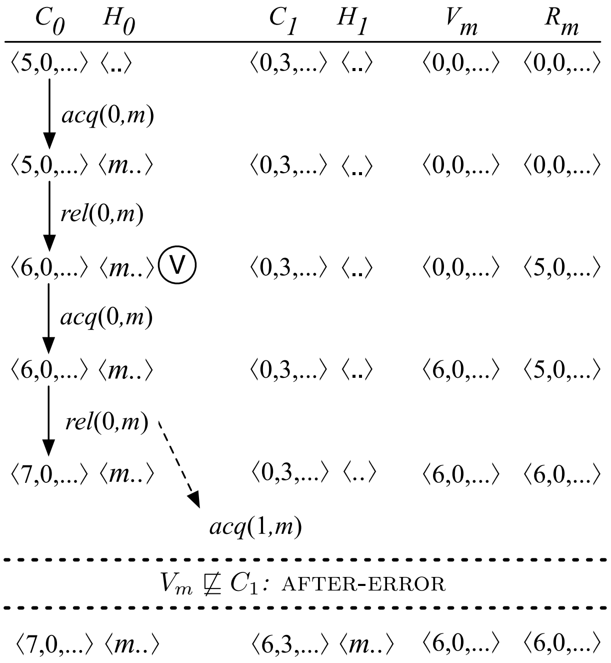

C_0 H_0 C_1 H_1 V_m R_m
⟨5,0,...⟩ ⟨..⟩ ⟨0,3,...⟩ ⟨..⟩ ⟨0,0,...⟩ ⟨0,0,...⟩
acq(0,m)
⟨5,0,...⟩ ⟨m..⟩ ⟨0,3,...⟩ ⟨..⟩ ⟨0,0,...⟩ ⟨0,0,...⟩
rel(0,m)
⟨6,0,...⟩ ⟨m..⟩ V ⟨0,3,...⟩ ⟨..⟩ ⟨0,0,...⟩ ⟨5,0,...⟩
acq(0,m)
⟨6,0,...⟩ ⟨m..⟩ ⟨0,3,...⟩ ⟨..⟩ ⟨6,0,...⟩ ⟨5,0,...⟩
rel(0,m)
⟨7,0,...⟩ ⟨m..⟩ ⟨0,3,...⟩ ⟨..⟩ ⟨6,0,...⟩ ⟨6,0,...⟩
acq(1,m)
$V_m \not\sqsubseteq C_1$: AFTER-ERROR
⟨7,0,...⟩ ⟨m..⟩ ⟨6,3,...⟩ ⟨m..⟩ ⟨6,0,...⟩ ⟨6,0,...⟩

Figure 7: Illustration of the Analysis State When Discovering an AFTER-ERROR

Checks indicate when the analysis should flag an error. Updates describe how the next analysis state is obtained.

For example, the rule [FIRST ACQUIRE] has two branches with negated counterparts in different rules: $H_t \neq$ `NotInX` and $m \notin H_t$, and one internal branch: if $A_m \sqsubseteq C_t$.... The crucial check is $V_m \sqsubseteq C_t$. If this check is violated, SIDETRACK issues an AFTER-ERROR warning.

Blame assignment.

Our analysis as presented is relatively straightforward to implement, and successfully flags feasible atomicity violations. Accurate blame assignment requires significant extra work, especially since BEFORE-ERROR and AFTER-ERROR atomicity violations do not manifest in the observed trace.

In the presence of nested transactions, it is not sufficient to report the (inner) transaction where the second acquire of a vulnerable window occurs. To illustrate this difficulty, consider the program shown in Figure 9, where we assume that all methods should be atomic.

While methods `c` and `d` are atomic, `a` and `b` potentially have atomicity violations. When we encounter the second acquire in the vulnerable window inside of `b`, the call stack may look like `a:b:c:d`. However this call stack is not precise enough to tell us which method is non-atomic. In fact, we need the call stack for the first acquire of m inside the vulnerable window as well: `a:b:d`. We need to identify the common prefix in the two stacks, and report all methods in this prefix as non-atomic (here, `a` and `b`). We store a call stack for the first and second acquires for the last vulnerable window in the instrumentation state associated with each lock, and call stacks for all locks acquired in the current transaction in the instrumentation state associated with each thread. With a suitable immutable linked-list representation, recording such a stack is a constant-time operation.

Thread 1	*Thread 2*

```
                          sync(m) {
                            sync(n) {...}
                          }
atomic {
  sync(n) {
    sync(m) {...}
    sync(m) {...}
  }
}
```

Thread 1	*Thread 2*

```
atomic {
  sync(n) {
    sync(m) {...}
    sync(m) {...}
  }
}
                          sync(m) {
                            sync(n) {...}
                          }
```

Figure 8: SIDETRACK flags the trace on the bottom with an atomicity violation, but not the trace on the top.

Tool complementation.

We can combine SIDETRACK with VELODROME [17], a sound and complete dynamic atomicity analysis, to increase coverage. VELODROME finds more atomicity violations from a particular trace than SIDETRACK, since VELODROME also looks for atomicity violations involving accesses to shared variables as well as locks. However, SIDETRACK finds feasible errors that VELODROME misses, since the latter does not generalize its analysis to other feasible traces. Thus the two tools are compatible and complementary. Since ROADRUNNER supports tool composition, it is straightforward to run these two together.

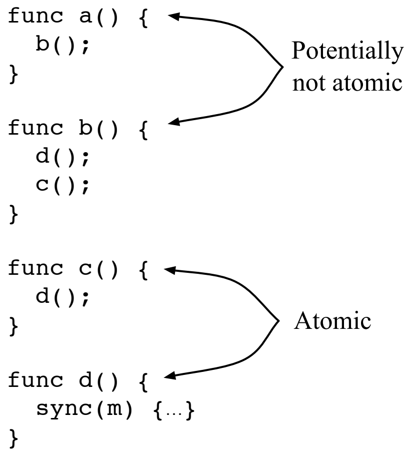
Test case generation.

SIDETRACK was tested using Tiddle [26], a trace description language and compiler that translates trace fragments into deterministic multithreaded Java programs. We have found that it is substantially easier to develop a custom test suite tailored for a particular dynamic analysis using Tiddle, as describing "good" and "bad" trace scenarios is simply a matter of jotting down the appropriate trace fragments.

5. EVALUATION

We present encouraging experimental results for SIDETRACK. We ran these experiments on a machine with 3 GB memory, 2.16 GHz dual-core CPU, running Mac OS X 10.5.6, and with Java HotSpot 64-Bit Server JVM 1.6.0.

Our benchmark set includes: `elevator`, a real-time discrete event simulator [30]; `colt`, a library for high performance scientific computing [5]; `jbb`, the SPEC JBB2000 simulator for business middleware [28]; `hedc`, a warehouse webcrawler for astrophysics data [30]; `barrier`, a barrier synchronization performance benchmark [19]; `philo`, a dining philosophers application [9]; `tsp`, a solver for the traveling salesman problem [30]; and `sync`, a synchronization performance benchmark [19]. Benchmarks from Java Grande [19]



```
func a() {
  b();
}

func b() {
  d();
  c();
}

func c() {
  d();
}

func d() {
  sync(m) {...}
}
```

Figure 9: Program Example Where Precise Blame Assignment Is Necessary

Table 1: Atomicity Errors Found by SideTrack.

Programs	Methods with Errors	Types of Errors			Predicted befores	Predicted afters	Predicted Total
		before-errors	in-errors	after-errors			
elevator	5	3	1	5	2	4	4
colt	9	4	7	9	2	2	2
jbb	10	7	5	10	5	5	5
hedc	4	1	4	4	0	0	0
barrier	1	1	1	1	0	0	0
philo	1	1	1	1	0	0	0
tsp	4	4	4	4	0	0	0
sync	4	4	4	4	0	0	0
Total	38	25	27	38	9	11	11

Table 2: Benchmark Performance Results

Programs	Num Threads	LOC	Base Runtime (Seconds)	Slowdown (Empty)	Slowdown (SideTrack)
colt	11	111,421	16.2	1.2	1.2
barrier	4	774	55.2	1.0	1.0
tsp	5	706	1.1	2.6	3.7
sync	4	650	68.8	0.8	0.9
crypt	7	1,241	0.6	3.9	4.3
moldyn	4	1,402	1.7	3.1	4.2
forkjoin	187	591	0.04	45.0	47.0
lufact	4	1,627	0.3	8.8	9.4
montecarlo	4	3,669	2.4	2.3	2.6
raytracer	4	1,970	1.5	5.4	13.9
series	4	967	2.9	1.5	1.8
sor	4	1,005	0.3	5.8	7.0

were configured to use 4 threads and the base data set. This set totals over 200K lines of code. Excluding the Java standard libraries, all classes loaded by benchmark programs were instrumented.

To check for atomicity violations, we assumed that *all methods should be atomic.* In practice this assumption works fairly well; previous experiments have validated that atomicity is a fundamental design principle for concurrency [13, 21].

For each benchmark, Table 1 reports both the total number of methods with atomicity violations (Column 2) as well as the error count for each type of atomicity violation (Columns 3 - 5). Note that there is significant overlap between the methods reported by each kind of violation. To further clarify the benefit of SideTrack's predictive analysis, Column 6 reports the number of before-errors that are not in-errors (and so not detected by earlier precise tools), and Column 7 reports on after-errors that are not in-errors. Finally, the last column reports on errors that are either before-errors or after-errors, but not in-errors, and so most clearly summarizes the improvement achieved via predictive analysis. SideTrack's predictive ability catches an additional 11 errors, in addition to the 27 in-errors: an improvement of roughly 40%.

Interestingly, all 11 of these additional violations were found by the *after* analysis, which suggests that the *after* analysis generalizes better than the *before* analysis.

To confirm that the predicted errors are not false positives, we investigated the error messages in the "Predicted Total" column by inspecting the program source code. We found that it was easy to pinpoint the errors with the blame assignment information. All were judged to be vulnerable to atomicity violations. However, some of these violations may be benign.

Table 2 reports on the performance of our analysis. For each of the compute-bound benchmarks, we report on the running time of that benchmark (without any instrumentation), and on the slowdown incurred by RoadRunner when running with the Empty tool (which just measures the instrumentation overhead but performs no dynamic analysis); and the slowdown when run with SideTrack. We also measured performance for all other benchmarks in the Java Grande suite; SideTrack reported no errors (of any of the three types) for these additional benchmarks. The average slowdown for SideTrack was 8.1x, as compared with a slowdown for the Empty tool of 6.8x.

There are a couple outliers. The `forkjoin` benchmark taxes our instrumentation framework by forking and joining almost 200 threads, each of which has an associated instrumentation state. Additionally, the running time of this benchmark is so short that other effects (like printing) may dominate the slowdown. Without `forkjoin`, the slowdown for SideTrack is 4.5x and the slowdown for the Empty tool is 3.3x. The 13.9x slowdown for `raytracer` is a result of the large number of small method calls in this benchmark, each of which must be recorded for blame assignment.

The results show that SideTrack provides a significant improvement in performance of prior dynamic atomicity analyses, such as SingleTrack [25] (10.4x), Velodrome [17] (10.3x), and Atomizer [13]. This performance improvement is largely because SideTrack does not analyze memory reads and writes, and instead assumes that the target program is race-free and any inter-thread communication is mediated via synchronization idioms such as locks. If necessary, this race-free assumption can be verified by concurrently running an efficient race detector such as FastTrack [14], which would incur an additional performance overhead of 10x. We believe using separate analyses to verify race-freedom and atomicity offers benefits both in terms of performance and modularity of the analysis code.

6. RELATED WORK

Atomicity violation detection tools.

A variety of tools have been developed to detect atomicity violations, both statically and dynamically. Static analyses for verifying atomicity include type systems [18, 15, 27, 16, 32] as well as techniques that look for cycles in the happens-before graph [10]. Compared to dynamic techniques, static systems provide stronger soundness guarantees but typically involve trade-offs between precision and scalability.

Dynamic techniques analyze a specific executed trace at runtime. Artho *et al.* [2] have developed a dynamic analysis tool to identify one class of "higher-level races" using the notion of view consistency. Many dynamic techniques for detecting atomicity violations exist in the literature; we will expand on some of them here. The Atomizer [13] uses Lipton's theory of reduction [20] to check serializability, similar to the reduction-based algorithms described in [31, 34]. Atomizer looks for irreducible patterns in the trace and may find some errors that do not actually manifest in the observed trace. However, Atomizer may also report false positives, due to limitations in how Atomizer reasons about different synchronization idioms. In contrast, Velodrome [17] reports an error if and only if the observed trace is not serializable. Velodrome uses transactional happens-before graphs to detect cycles, which correspond directly to an atomicity violation. Farzan and Madhusudan [11] provide space complexity bounds for a similar analysis.

Predictive approaches.

There have been a number of recent papers about predicting atomicity violations from an observed trace; note that none of these approaches are completely online. However, some of these tools offer increased coverage over SideTrack by using additional static information.

HAVE [7] injects static method summaries into a dynamic context and speculates on unexecuted control flow branches to predict atomicity violations. This speculative technique may lead to false positives, but experimental results indicate that such messages are very rare. The kinds of predictions made by HAVE are different from those made by SideTrack, since HAVE bases its predictions on unobserved-yet-feasible operations, while SideTrack uses the observed trace to infer a different-but-feasible scheduling.

JPredictor [6] performs offline causality slicing on an observed trace to remove irrelevant events, then generates sound permutations of events to predict atomicity violations. Atomicity violations are detected by matching against 11 known violation patterns [29]. The hybrid techniques of HAVE and JPredictor make use of static control- and data-flow information and thus are complementary to our purely dynamic prediction technique.

Wang and Stoller have developed a block-based algorithm [34] and commit-node algorithms that address both conflict-atomicity (referred to as atomicity in this paper) and view-atomicity [33]. They monitor execution traces at runtime, and analyze those traces offline. By design, these algorithms report potential serializability violations that do not occur on the observed trace but which might occur on other traces. Although these algorithms potentially report false positives, this problem does not seem to arise in practice.

Farzan and Madhusudan [12] predict runs from program models based on *profiles* and check serializability of the predicted runs. They develop time bounds and algorithms for straight-line, regular, and recursive programs with and without synchronization. Implementing prediction tools for atomicity violations based on their theoretical results remains future work.

Scheduling atomicity violations.

Sen and Park [24] have developed a tool called AtomFuzzer which attempts to schedule atomicity violations. They target the same pattern we do: one thread acquires the same lock twice within a transaction. They use the heuristic that every synchronized block should be atomic, instead of every method. When a thread is about to acquire the same lock a second time, AtomFuzzer pauses the thread (with probability 0.5), and tries to schedule another thread (which may cause an atomicity violation) first. If all threads are paused, one thread is chosen to continue.

AtomFuzzer may miss some errors which we are able to catch, because it is not always possible (or desirable) to pause every thread as long as necessary at a vulnerable point. However, since AtomFuzzer actually produces an atomicity violation, the user is able to see what sorts of problems (e.g. exceptions) are caused by that violation. Additionally, AtomFuzzer may find errors that we are unable to report, by shaping a particular execution trace through scheduling. For example, AtomFuzzer may be able to drive execution to produce an atomicity violation for the trace in Figure 5, whereas we cannot discover if such a trace is feasible. These two approaches are complementary; scheduling (also see [8]) can be used to generate more atomicity violations for a particular trace while an analysis like SideTrack predicts other feasible violations.

7. FUTURE WORK

Several issues remain for future work. We are currently developing formal proofs of our intended correctness property, namely that the analysis is complete and reports no false alarms on data-race-free programs. We are also extending our analysis to handle accesses to volatile variables and other synchronization idioms, such as barriers, wait-notify, etc. We would also like to extend this general approach for trace generalization to other specification idioms, such as race freedom and deterministic parallelism [25, 4].

8. REFERENCES

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[2] C. Artho, K. Havelund, and A. Biere. High-level data races. In *International Workshop on Verification and Validation of Enterprise Information Systems*, 2003.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[4] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for Deterministic Parallel Java. Technical Report UIUCDCS-R-2009-3032, Department of Computer Science, University of Illinois at Urbana-Champaign, 2009.

[5] CERN. Colt 1.2.0. Available from `http://dsd.lbl.gov/~hoschek/colt/`, 2007.

[6] F. Chen, T. F. Şerbănuţă, and G. Roşu. jPredictor: a predictive runtime analysis tool for Java. In *International Conference on Software Engineering*, 221–230. ACM, 2008.

[7] Q. Chen, L. Wang, Z. Yang, and S. D. Stoller. HAVE: Integrated dynamic and static analysis for atomicity violations. In *International Conference on Fundamental Approaches to Software Engineering (FASE)*, 2009.

[8] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded java programs. In *Concurrency and Computation: Practice and Experience*, volume 15, 2003.

[9] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware Java runtime. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 245–255, 2007.

[10] A. Farzan and P. Madhusudan. Causal atomicity. In *Computer Aided Verification (CAV)*, 315–328, 2006.

[11] A. Farzan and P. Madhusudan. Monitoring atomicity in concurrent programs. In *Computer Aided Verification (CAV)*, 52–65, 2008.

[12] A. Farzan and P. Madhusudan. The complexity of predicting atomicity violations. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2009.

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

[14] C. Flanagan and S. N. Freund. FastTrack: Efficient and precise dynamic race detection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009. To appear.

[15] C. Flanagan, S. N. Freund, M. Lifshin, and S. Qadeer. Types for atomicity: Static checking and inference for Java. *Transactions on Programming Languages and Systems*, 30(4):1–53, 2008.

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

[17] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[18] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 338–349, 2003.

[19] Java Grande Forum. Java Grande benchmark suite. `http://www.javagrande.org`, 2008.

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

[21] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *SIGPLAN Notices*, 2008.

[22] F. Mattern. Virtual time and global states of distributed systems. In *International Workshop on Parallel and Distributed Algorithms*. 1988.

[23] R. H. B. Netzer and B. P. Miller. What are race conditions? Some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1:74–88, 1992.

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

[25] C. Sadowski, S. N. Freund, and C. Flanagan. SingleTrack: A dynamic determinism checker for multithreaded programs. In *European Symposium on Programming (ESOP)*, 2009.

[26] C. Sadowski and J. Yi. Tiddle: A trace description language for generating concurrent benchmarks to test dynamic analyses. In *International Workshop on Dynamic Analysis (WODA)*, 2009. To appear.

[27] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller. Automated type-based analysis of data races and atomicity. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 83–94, 2005.

[28] SPEC. Standard Performance Evaluation Corporation JBB2000 Benchmark. `http://www.spec.org/osg/jbb2000/`, 2000.

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

[30] C. von Praun and T. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 115–128, 2003.

[31] L. Wang and S. D. Stoller. Runtime analysis for atomicity. In *International Workshop on Runtime Verification (RV)*, 2003.

[32] L. Wang and S. D. Stoller. Static analysis of atomicity for programs with non-blocking synchronization. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 61–71, 2005.

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

[34] L. Wang and S. D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Transactions on Software Engineering*, 32:93–110, Feb. 2006.