Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs

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

Stephen N. Freund
Computer Science Department
Williams College
Williamstown, MA 01267

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

Abstract

Atomicity is a fundamental correctness property in multithreaded programs, both because atomic code blocks are amenable to sequential reasoning (which significantly simplifies correctness arguments), and because atomicity violations often reveal defects in a program's synchronization structure. Unfortunately, all atomicity analyses developed to date are *incomplete* in that they may yield false alarms on correctly synchronized programs, which limits their usefulness.

We present the first dynamic analysis for atomicity that is both sound and complete. The analysis reasons about the exact dependencies between operations in the observed trace of the target program, and it reports error messages if and only if the observed trace is not conflict-serializable. Despite this significant increase in precision, the performance and coverage of our analysis is competitive with earlier incomplete dynamic analyses for atomicity.

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

General Terms Languages, Algorithms, Verification

Keywords Atomicity, serializability, dynamic analysis

1. Introduction

Reasoning about the behavior and correctness of multithreaded programs is notoriously difficult, due to both memory-model issues and the non-deterministic interleaving of the various threads. The widespread adoption of multicore processors and increasingly-multithreaded software significantly exacerbates this reliability problem. Indeed, the advent of multi-core processors may actually degrade the reliability of our software infrastructure. To avoid this undesirable outcome, better tools for building reliable concurrent systems are clearly needed.

For most multithreaded programs, an important first step is to verify the key correctness properties of race-freedom and atomicity.

PLDI'08, June 7–13, 2008, Tucson, Arizona, USA.

- Race-freedom guarantees that the program's behavior can be understood as if it executes on a sequentially-consistent memory model [17].
- Atomicity guarantees that the program's behavior can be understood as if each atomic block executes serially (without interleaved steps of other threads), which enables straightforward sequential reasoning.

Race-freedom and atomicity are complementary properties. For example, the following method `Set.add` is free of race conditions because all shared mutable variables are correctly synchronized. However, `Set.add` still violates its atomicity specification because other threads could update the underlying vector `elems` between the calls to `elems.contains` and `elems.add`.

```
class Set {
  final Vector elems = new Vector();
  atomic void add(Object x) {
    if (!elems.contains(x)) elems.add(x);
  }
}
class Vector {
  synchronized void add(Object o) { ... }
  synchronized boolean contains(Object o) { ... }
}
```

For race conditions, a variety of dynamic race detection tools have been developed [36, 33, 47], including complete race detectors that never produce false alarms [34, 37, 7].

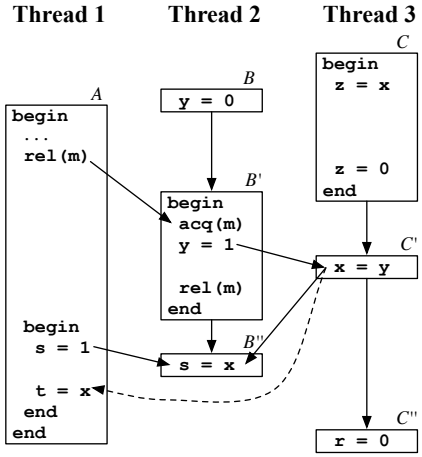
For atomicity, a variety of dynamic analyses have also been developed (e.g. [11, 44, 1, 43, 46]), but these tools are all incomplete in that they report false alarms on some correctly-synchronized programs. For example, the Atomizer [11] uses the Lockset algorithm [36] to reason about standard mutual-exclusion locks, but may report false alarms if the target program uses additional synchronization idioms to protect its data.

Indeed, it has proven surprisingly difficult and time consuming to identify real errors among the spurious warnings produced by these tools. Even if a code block looks suspicious, it may still be atomic due to some subtle synchronization discipline that is not (yet) understood by the current programmer or code maintainer. Additional real bugs (*e.g.*, deadlocks) could be easily introduced while attempting to "fix" a spurious warning produced by these tools. Conversely, real errors could be ignored because they appear to be false alarms.

Sound and Complete Atomicity Checking. To address these problems, this paper presents the first dynamic analysis for atomicity that is both *sound* (it reports an error if the observed trace is non-serializable) and *complete* (it reports an error only for

non-serializable traces). An execution trace is considered *serializable* (also referred to as *conflict-serializable*) if it can be transformed into an equivalent serial trace by commuting adjacent, non-conflicting operations of different threads [4].

We illustrate the behavior of our analysis in terms of the following trace diagram, where the vertical ordering of instructions reflects their interleaved execution. The operations `begin` and `end` demarcate *atomic blocks*, which are intended to be serializable. Atomic blocks may be nested (due to function calls, etc), and the outermost block starts a new *transaction*. Transactions are indicated via boxes with associated labels A, B, B', etc. Operations outside an atomic block execute in their own unary transaction. The primary goal of our analysis is to verify that each transaction (and thus each atomic block) in the observed trace is serializable.



Our analysis infers *happens-before* edges [26] between program operations. These edges reflect synchronization, communication, and thread ordering, as shown via arrows in the above diagram. We then lift this relation from operations to a *transactional happens-before relation* $\lessdot$. For the above trace, our analysis infers that $A \lessdot B'$ (via the release-acquire edge on `m`), $B' \lessdot C'$ (via the write-read edge on `y`), and finally that $C' \lessdot A$ (via the write-read edge on `x`). At this stage, our analysis detects a cycle $A \lessdot B' \lessdot C' \lessdot A$ in the transactional happens-before graph, which reveals that the observed trace is not serializable.[1]

Blame Assignment. Given the difficulty in understanding the warnings produced by prior tools, a particular focus of this work is on *blame assignment*, *e.g.* how to further localize the error in the above cycle $A \lessdot B' \lessdot C' \lessdot A$. In particular, our analysis detects that this cycle reflects the following happens-before path on operations:

$$A{:}\texttt{rel(m)} \;<\; B'{:}\texttt{acq(m)} \;<\; B'{:}\texttt{y=1} \;<\; C'{:}\texttt{x=y} \;<\; A{:}\texttt{t=x}$$

Since the above path interleaves transaction A with other conflicting operations, there is no equivalent trace where A executes serially, and so we blame transaction A for this atomicity violation. Furthermore, only the outermost atomic block in A is blamed; no error is reported for the inner block, which is serializable.

Transactional Happens-Before Representation. A key contribution of this work is an efficient and scalable representation of the transactional happens-before relation. The traditional representation technique of clock vectors [30] is not applicable because our happens-before relation is over compound transactions and not individual operations. Moreover, a trace may contain millions of transactions, and storing the entire happens-before graph would be infeasible. Our analysis uses two techniques to avoid this:

- The analysis garbage collects transactions as soon as it can guarantee they will never occur on a cycle. In the above graph, transactions B and C are collected as soon as they terminate, but transactions B', B'', C', and C'' are kept alive until A terminates.
- The analysis dynamically detects when transaction allocation is not even necessary. For example, since transaction B would be collected as soon as it terminates, it is not allocated in the first place, and transaction C'' is merged with its predecessor C'.

Empirical Validation. Combining the above techniques with careful data-representation choices yields a sound and complete atomicity analysis whose performance is competitive with earlier atomicity analyses, and which provides a significant increase in accuracy. On a range of benchmarks, the Atomizer [11] detects 154 non-atomic methods but produces 84 false alarms. In contrast, Velodrome (our prototype checker for Java) detects 133 non-atomic methods (and misses the remaining 21 by not generalizing from the observed traces), but produces zero false alarms. Thus, in the terminology of information retrieval, Velodrome provides slightly less recall but vastly increased precision — a trade-off that we believe is quite appropriate for all but the most safety-critical applications.

In addition, Velodrome and the Atomizer are complementary tools. A promising approach is to run both tools simultaneously, correcting real Velodrome-reported errors as a top priority, and investigating the additional Atomizer-reported warnings as a lower priority task.

We also enhance coverage by extending our analysis to explore interleavings that are more likely to be non-serializable. In particular, we use the Atomizer to recognize potential atomicity violations, such as an unsynchronized read-modify-write sequence. Our tool then temporarily blocks the thread performing those operations in the hope that an interleaved write by a second thread will provide a concrete witness to the violation. Preliminary experience with this technique is quite promising. It enabled us to find several additional atomicity violations in our benchmarks, and it substantially improved Velodrome's success rate at finding randomly-inserted synchronization defects in several small programs.

Summary. This paper makes the following main contributions:

- We present the first dynamic analysis for atomicity that is both sound and complete: the analysis identifies exactly those traces that are not conflict-serializable.
- The analysis performs precise blame assignment, and can typically assign blame for each violation to particular instructions in a particular atomic block.
- We show that completeness can be achieved with little additional overhead: the performance of our analysis is competitive with earlier dynamic analyses.
- On a range of standard benchmarks, we showed that our analysis detects almost all (85%) non-atomic methods detected by the Atomizer, while reducing the false alarm ratio from roughly 40% to 0%.
- We leverage the Atomizer's dynamic analysis to heuristically guide our checker to explore traces more likely to exhibit atomicity violations. This technique provides increased coverage with no loss of completeness.

Outline. The following section formalizes the semantics of multithreaded programs. Section 3 describes an initial version of our analysis, and Section 4 refines it to achieve better performance and

[1] Similar ideas have been explored in the database literature [4]; our work adapts these ideas to general-purpose programming languages, in a similar spirit to much recent work on transactional memory.

more precise blame assignment. Sections 5 and 6 describe and evaluate our implementation, respectively. Sections 7 and 8 conclude with a discussion of related work and potential future work.

2. Atomicity in Multithreaded Programs

To provide a formal basis for our dynamic analysis, we first define the execution semantics of multithreaded programs in a style similar to [11]. A program consists of a number of concurrently executing threads, each of which has an associated thread identifier $t \in Tid$.[2] Each thread has its own local store π containing thread-local data, such as the program counter and call stack. In addition, the threads communicate through a global store σ, which is shared by all threads. The global store maps program variables x to values v. The global store also records the state of each lock variable $m \in Lock$. If $\sigma(m) = t$, then the lock m is held by thread t; if $\sigma(m) = \bot$, then that lock is not held by any thread.

A state $\Sigma = (\sigma, \Pi)$ of the multithreaded system consists of a global store σ and a mapping Π from thread identifiers t to the local store $\Pi(t)$ of each thread. Execution starts in an initial state $\Sigma_0 = (\sigma_0, \Pi_0)$.

Transitions. The behavior of each thread is captured via the transition relation $T \subseteq Tid \times LocalStore \times Operation \times LocalStore$. The relation $T(t, \pi, a, \pi')$ holds if the thread t can take a step from a local store π to a new local store π' by performing the operation $a \in Operation$ on the global store. The set of possible operations that a thread t can perform on the global store include:

- $rd(t, x, v)$ and $wr(t, x, v)$, which read a value v from variable x and write a value v to x, respectively.
- $acq(t, m)$ and $rel(t, m)$, which acquire and release a lock m.
- $begin^l(t)$ and $end(t)$, which mark the beginning and end of an atomic block. (The label l identifies a particular atomic block, and is used for error reporting.)

In code examples, we often omit t and l when they are clear from the context or irrelevant, and we use more familiar syntax, such as `x = v`, for reads and writes. We use the function $tid(a)$ to extract the thread identifier from an operation.

The relation $\sigma \rightarrow^a \sigma'$ models the effect of an operation a on the global store σ: see Figure 1. In these rules, the global store $\sigma[x := v]$ is identical to σ but maps the variable x to the value v.

The transition relation $\Sigma \rightarrow^a \Sigma'$ performs a single step of computation. It chooses an operation a by thread t that is applicable in the local store $\Pi(t)$, performs that operation on the current global store σ yielding a new global store σ', and returns a new state $(\sigma', \Pi[t := \pi'])$ containing the new global and local stores σ' and π'.

Traces. A *trace* α is a sequence of operations that captures an execution of a multithreaded program by describing the operations performed by each thread and their ordering. The behavior of a trace $\alpha = a_1.a_2.\cdots.a_n$ is defined by the relation $\Sigma_0 \rightarrow^\alpha \Sigma_n$, which holds if there exist intermediate states $\Sigma_1, \ldots, \Sigma_{n-1}$ such that $\Sigma_0 \rightarrow^{a_1} \Sigma_1 \rightarrow^{a_2} \cdots \rightarrow^{a_n} \Sigma_n$.

A *transaction* in a trace α is the sequence of operations executed by a thread t starting with a $begin^l(t)$ operation and containing all t operations up to and including a matching $end(t)$ operation, or up to the end of the trace, if there is no matching $end(t)$ operation. In addition, if an operation a by thread t does not occur within an atomic block for t, then the operation a by itself is considered a (unary) transaction. Thus, each transaction is non-empty.

[2] Although dynamic thread creation is not explicitly supported by the semantics, it can be modeled within the semantics in a straightforward way.

Figure 1: Semantics of Multithreaded Programs

Domains:

$u, t \in Tid$ $\quad x \in Var$ $\quad v \in Value$
$m \in Lock$ $\quad \pi \in LocalStore$

$$\Pi \in LocalStores = Tid \rightarrow LocalStore$$
$$\sigma \in GlobalStore = (Var \rightarrow Value) \cup (Lock \rightarrow (Tid \cup \{\bot\}))$$
$$\Sigma \in State = GlobalStore \times LocalStores$$

Operations:

$$a \in Operation ::= rd(t,x,v) \mid wr(t,x,v) \mid acq(t,m) \mid rel(t,m) \mid begin^l(t) \mid end(t)$$
$$l \in Label$$

Effect of operations: $\sigma \rightarrow^a \sigma'$

[ACT READ]
$$\frac{\sigma(x) = v}{\sigma \rightarrow^{rd(t,x,v)} \sigma}$$

[ACT WRITE]
$$\frac{}{\sigma \rightarrow^{wr(t,x,v)} \sigma[x := v]}$$

[ACT ACQUIRE]
$$\frac{\sigma(m) = \bot}{\sigma \rightarrow^{acq(t,m)} \sigma[m := t]}$$

[ACT RELEASE]
$$\frac{\sigma(m) = t}{\sigma \rightarrow^{rel(t,m)} \sigma[m := \bot]}$$

[ACT OTHER]
$$\frac{a \in \{begin^l(t), end(t)\}}{\sigma \rightarrow^a \sigma}$$

State transition relation: $\Sigma \rightarrow^a \Sigma'$

[STD STEP]
$$\frac{t = tid(a) \qquad T(t, \Pi(t), a, \pi') \qquad \sigma \rightarrow^a \sigma'}{(\sigma, \Pi) \rightarrow^a (\sigma', \Pi[t := \pi'])}$$

Serializable Traces. A trace is *serial* if each transaction's operations execute contiguously, without interleaved operations of other threads. The notion of serializability is based on the idea of conflicting operations. Two operations in a trace *conflict* if:

1. they access (read or write) the same variable, and at least one of the accesses is a write;
2. they operate on (acquire or release) the same lock; or
3. they are performed by the same thread.

If two operations do not conflict, they *commute*. Two traces are *equivalent* if one can be obtained from the other by repeatedly swapping adjacent commuting operations. Equivalent traces exhibit equivalent behavior. A trace is *serializable* if it is equivalent to some serial trace.

Examples. In the following trace, the read-modify-write sequence of Thread 1 is interleaved with a write by Thread 2. This trace is clearly not serial; it is also not serializable, because the write by Thread 2 conflicts with both the read and write by Thread 1 and cannot be commuted outside the atomic block.

Thread 1	Thread 2
`begin`	
`  tmp = x`	
	`x = 0`
`  x = tmp + 1`	
`end`	

Atomicity violations such as this one can be caught by the Atomizer [11] and other dynamic atomicity tools [44, 43], but these tools are prone to false alarms. For example, the Atomizer uses Eraser's LockSet algorithm [36] to reason about lock-based synchronization and cannot understand more complex synchronization patterns. To illustrate this limitation, the following program uses a volatile variable `b` to indicate whether thread 1 or thread 2 has exclusive access to the shared variable `x`.

Thread 1	**Thread 2**
`while (true) {`	`while (true) {`
`  while (b != 1) {`	`  while (b != 2) {`
`  skip;`	`  skip;`
`  }`	`  }`
`  begin`	`  begin`
`  int tmp = x;`	`  int tmp = x;`
`  x = tmp + 1;`	`  x = tmp + 1;`
`  b = 2;`	`  b = 1;`
`  end`	`  end`
`}`	`}`

Even though this program yields only serializable traces, the Atomizer will report false alarms because it cannot understand the program's synchronization discipline; other atomicity tools behave in a similar fashion.

3. Dynamic Analysis for Serializability

We now describe our dynamic analysis for precisely identifying non-serializable traces. Given a trace α, the *happens-before* relation $<_\alpha$ for α is the smallest transitively-closed relation on operations such that if an operation a occurs before b in α, and a conflicts with b, then a *happens-before* b.[3] The transactional structure of traces induces an equivalence relation on operations: $a \sim_\alpha b$ if a and b occur in the same transaction in α. Since all operations in a transaction are intended to (conceptually) happen contiguously, we combine these two relations into a (transitively-closed) *extended happens-before relation*:

$$\lessdot\!\sim_\alpha \stackrel{\text{def}}{=} (<_\alpha \cup \sim_\alpha)^*$$

We lift this extended happens-before relation from operations to transactions, and so a transaction A *happens-before* transaction B in α (written $A \lessdot_\alpha B$) if $A \neq B$ and there exists some operations a of A and b of B such that $a \lessdot\!\sim_\alpha b$. We then leverage existing results in database theory [4] to show that α is serializable *if and only if* the transactional happens-before order $\lessdot_\alpha$ is acyclic.

Analysis Details. Our analysis is an online algorithm that maintains an *analysis state* ϕ; when the target program performs an operation a, the analysis updates its state via the relation $\phi \Rightarrow^a \phi'$.

For clarity, we initially present a basic version of our analysis. This initial analysis allocates a node in the happens-before graph for each transaction in the observed trace. We let $Node$ denote the set of such nodes, and $Node_\bot = Node \cup \{\bot\}$. The instrumentation store $\phi = (\mathcal{C}, \mathcal{L}, \mathcal{U}, \mathcal{R}, \mathcal{W}, \mathcal{H})$ is a tuple of six components:

- $\mathcal{C} : Tid \to Node_\bot$ identifies the current transaction node (if any) for each thread;
- $\mathcal{L} : Tid \to Node_\bot$ identifies the transaction that executed the last operation (if any) of each thread;
- $\mathcal{U} : Lock \to Node_\bot$ identifies the last transaction (if any) to release or unlock each lock;
- $\mathcal{R} : Var \times Tid \to Node_\bot$ identifies the last transaction of each thread to read from each variable;
- $\mathcal{W} : Var \to Node_\bot$ identifies the last transaction (if any) to write to each variable; and
- $\mathcal{H} \subseteq Node \times Node$ is the happens-before relation on transactions. (More precisely, the transitive closure $\mathcal{H}^*$ of $\mathcal{H}$ is the happens-before relation, since, for efficiency, $\mathcal{H}$ is not transitively closed.)

[3] In theory, a particular operation a could occur multiple times in a trace. We avoid this complication by assuming that each operation includes a unique identifier, but, to avoid clutter, we do not include this unique identifier in the concrete syntax of operations.

Figure 2: Instrumentation Relation: $\phi \Rightarrow^a \phi$

[INS ENTER]
$$\frac{\mathcal{C}(t) = \bot \qquad \text{n is fresh} \qquad \mathcal{H}' = \mathcal{H} \uplus \{(\mathcal{L}(t), n)\} \qquad \mathcal{C}' = \mathcal{C}[t := n]}{(\mathcal{C}, \mathcal{L}, \mathcal{U}, \mathcal{R}, \mathcal{W}, \mathcal{H}) \Rightarrow^{begin^l(t)} (\mathcal{C}', \mathcal{L}, \mathcal{U}, \mathcal{R}, \mathcal{W}, \mathcal{H}')}$$

[INS EXIT]
$$\frac{n = \mathcal{C}(t) \qquad n \neq \bot \qquad \mathcal{C}' = \mathcal{C}[t := \bot] \qquad \mathcal{L}' = \mathcal{L}[t := n]}{(\mathcal{C}, \mathcal{L}, \mathcal{U}, \mathcal{R}, \mathcal{W}, \mathcal{H}) \Rightarrow^{end(t)} (\mathcal{C}', \mathcal{L}', \mathcal{U}, \mathcal{R}, \mathcal{W}, \mathcal{H})}$$

[INS ACQUIRE]
$$\frac{n = \mathcal{C}(t) \qquad n \neq \bot \qquad \mathcal{H}' = \mathcal{H} \uplus \{(\mathcal{U}(m), n)\}}{(\mathcal{C}, \mathcal{L}, \mathcal{U}, \mathcal{R}, \mathcal{W}, \mathcal{H}) \Rightarrow^{acq(t,m)} (\mathcal{C}, \mathcal{L}, \mathcal{U}, \mathcal{R}, \mathcal{W}, \mathcal{H}')}$$

[INS RELEASE]
$$\frac{n = \mathcal{C}(t) \qquad n \neq \bot \qquad \mathcal{U}' = \mathcal{U}[m := n]}{(\mathcal{C}, \mathcal{L}, \mathcal{U}, \mathcal{R}, \mathcal{W}, \mathcal{H}) \Rightarrow^{rel(t,m)} (\mathcal{C}, \mathcal{L}, \mathcal{U}', \mathcal{R}, \mathcal{W}, \mathcal{H})}$$

[INS READ]
$$\frac{n = \mathcal{C}(t) \qquad n \neq \bot \qquad \mathcal{R}' = \mathcal{R}[(x,t) := n] \qquad \mathcal{H}' = \mathcal{H} \uplus \{(\mathcal{W}(x), n)\}}{(\mathcal{C}, \mathcal{L}, \mathcal{U}, \mathcal{R}, \mathcal{W}, \mathcal{H}) \Rightarrow^{rd(t,x,v)} (\mathcal{C}, \mathcal{L}, \mathcal{U}, \mathcal{R}', \mathcal{W}, \mathcal{H}')}$$

[INS WRITE]
$$\frac{n = \mathcal{C}(t) \qquad n \neq \bot \qquad \mathcal{W}' = \mathcal{W}[x := n] \qquad \mathcal{H}' = \mathcal{H} \uplus (\{(\mathcal{R}(x,t'), n) \mid t' \in Tid\} \cup \{(\mathcal{W}(x), n)\})}{(\mathcal{C}, \mathcal{L}, \mathcal{U}, \mathcal{R}, \mathcal{W}, \mathcal{H}) \Rightarrow^{wr(t,x,v)} (\mathcal{C}, \mathcal{L}, \mathcal{U}, \mathcal{R}, \mathcal{W}'\mathcal{H}')}$$

[INS OUTSIDE]
$$\frac{\mathcal{C}(t) = \bot \qquad l \text{ is a fresh label} \qquad a \in \{acq(t,m), rel(t,m), rd(t,x,v), wr(t,x,v)\} \qquad \phi \Rightarrow^{begin^l(t)} \phi_1 \qquad \phi_1 \Rightarrow^a \phi_2 \qquad \phi_2 \Rightarrow^{end(t)} \phi'}{\phi \Rightarrow^a \phi'}$$

In the initial analysis state ϕ_0, these components are all empty:

$$\phi_0 = (\lambda t.\bot,\ \lambda t.\bot,\ \lambda m.\bot,\ \lambda x, t.\bot,\ \lambda x.\bot,\ \emptyset)$$

The relation $\phi \Rightarrow^a \phi'$ shown in Figure 2 updates the analysis state appropriately for each operation a of the target program. The first rule [INS ENTER] handles a $begin^l(t)$ operation, which starts a new transaction by thread t. The rule checks that thread t is not already in a transaction (*i.e.*, $\mathcal{C}(t) = \bot$) and updates $\mathcal{C}$ to record that thread t is inside a fresh transaction n. (We defer handling nested atomic blocks to the following section.) This rule uses the operation $\mathcal{H} \uplus E$ to extend the happens-before graph with additional edges $E \subseteq Node_\bot \times Node_\bot$, filtering out self-edges and edges that start or end on $\bot$:

$$\mathcal{H} \uplus E \stackrel{\text{def}}{=} \mathcal{H} \cup \{(n_1, n_2) \in E \mid n_1 \neq n_2,\ n_1 \neq \bot,\ n_2 \neq \bot\}$$

Thus, in [INS ENTER], if $\mathcal{L}(t) = \bot$, then the happens-before graph is unchanged. Otherwise it is extended with an edge from the last transaction of thread t to the current transaction of t. The rule [INS EXIT] handles $end(t)$ simply by updating $\mathcal{C}(t)$ and $\mathcal{L}(t)$ appropriately. The rule [INS ACQUIRE] for a lock acquire $acq(t, m)$ updates the happens-before graph with an edge from the last release $\mathcal{U}(m)$ of that lock. Conversely, the rule [INS RELEASE] for a lock release $rel(t, m)$ updates $\mathcal{U}(m)$ with the current transaction n.

The rule [INS READ] for a read operation $rd(t, x, v)$ records (1) that the last read of the variable x by thread t occurs in the current transaction n, and (2) that the last write to x happens before the current transaction (since reads and writes conflict). The rule [INS WRITE] for a write operation $wr(t, x, v)$ records that the last write to x is by the current transaction n, and that all previous accesses (reads or writes) to x happen before n.

For operations outside the dynamic scope of any atomic block (and thus outside any existing transaction), the rule [INS OUTSIDE] enters a new transaction, performs that operation, and then exits that transaction. This rule is simple but inefficient; optimized variants are described in the following section.

We extend the relation $\phi \Rightarrow^a \phi'$ from operations to traces: the relation $\phi_0 \Rightarrow^\alpha \phi_n$ holds for a trace $\alpha = a_1.\cdots.a_n$ if there exist intermediate analysis states $\phi_1, \ldots, \phi_{n-1}$ such that:

$$\phi_0 \Rightarrow^{a_1} \phi_1 \Rightarrow^{a_2} \cdots \Rightarrow^{a_{n-1}} \phi_{n-1} \Rightarrow^{a_n} \phi_n$$

Correctness. The set *Error* denotes analysis states that contain a non-trivial cycle in the happens-before relation:

$$Error \stackrel{\text{def}}{=} \{(\mathcal{C}, \mathcal{L}, \mathcal{U}, \mathcal{R}, \mathcal{W}, \mathcal{H}) \mid \mathcal{H}^* \text{ contains a non-trivial cycle}\}$$

Our central correctness result is that the dynamic analysis is both sound and complete: it identifies exactly the non-serializable traces. That is, if $\phi_0 \Rightarrow^\alpha \phi$ then

$$\phi \in Error \text{ if and only if } \alpha \text{ is not serializable.}$$

This result follows from the following inductive invariant describing how particular properties of the trace α are represented in the analysis state ϕ. This correspondence relies on a mapping $f : \alpha \to Node$ from each operation a in α to a corresponding node in the happens-before graph representing the transaction in which a appears.

DEFINITION 1. *Given $\phi = (\mathcal{C}, \mathcal{L}, \mathcal{U}, \mathcal{R}, \mathcal{W}, \mathcal{H})$, the invariant $Inv(\alpha, \phi, f)$ is the conjunction of the following conditions, for all $t \in Tid$, $x \in Var$, $m \in Lock$, and $a, b \in \alpha$:*

1. *If t is in a transaction at the end of α and a is the last operation by t in α, then $\mathcal{C}(t) = f(a)$.*
2. *If t is* not *in a transaction at the end of α, then*
   - *$\mathcal{C}(t) = \bot$.*
   - *If the last operation by t in α is a, then $\mathcal{L}(t) = f(a)$; if there is no such operation, then $\mathcal{L}(t) = \bot$.*
3. *If the last write to x in α is a, then $\mathcal{W}(x) = f(a)$; if there is no such write, then $\mathcal{W}(x) = \bot$.*
4. *If the last read of x by t in α is a, then $\mathcal{R}(x, t) = f(a)$; if there is no such read, then $\mathcal{R}(x, t) = \bot$.*
5. *If the last release of m in α is a, then $\mathcal{U}(m) = f(a)$; if there is no such release, then $\mathcal{U}(m) = \bot$.*
6. *If $f(a) = f(b)$ then $a \sim_\alpha b$.*
7. *If $a <_\alpha b$ then $(f(a), f(b)) \in \mathcal{H}^*$.*
8. *If $(n_1, n_2) \in \mathcal{H}$ then there exists $a_1, a_2 \in \alpha$ such that $f(a_1) = n_1$, $f(a_2) = n_2$ and $a_1 <_\alpha a_2$.*

THEOREM 1. *Given $\phi_0 \Rightarrow^\alpha \phi$,*

$$\phi \in Error \Leftrightarrow \alpha \text{ is not serializable}$$

PROOF SKETCH: The invariant $\exists f.\ Inv(\alpha, \phi, f)$ holds via a proof by induction on the length of the trace α. Hence:

$\phi \in Error$
$\Leftrightarrow$ $\mathcal{H}^*$ contains a non-trivial cycle
$\Leftrightarrow$ $\lessapprox_\alpha$ contains a cycle with operations from different transactions
$\Leftrightarrow$ $\lessdot_\alpha$ contains a cycle
$\Leftrightarrow$ α is not serializable

where the last step follows from a standard argument [4].

4. Extensions and Optimizations

The analysis presented so far is correct but requires substantial improvement in order to scale to realistic programs.

4.1 Garbage Collection

A trace may include many millions of transactions, making storage of the entire happens-before graph on transactions infeasible. Hence, a key challenge is *garbage collecting* old, redundant nodes.

References to a particular node n can be stored in the various components of the analysis state, with the result that an outgoing edge from n can be added at any time (for example, via [INS ACQUIRE], etc). A careful reading of the instrumentation rules, however, reveals that *incoming* edges to a node can be added only by the thread executing that transaction. Thus, if a transaction n has already finished (*i.e.*, $n \notin Range(\mathcal{C})$), additional incoming edges will never be added to n. Hence, a finished node n with no incoming edges (*i.e.*, $n \notin Range(\mathcal{H})$) will never occur on a cycle.

In this situation, we can safely garbage collect n and remove it from the happens-before graph. There still may be references to n from the analysis state components $\mathcal{L}, \mathcal{U}, \mathcal{W}$, and $\mathcal{R}$, but these are *weak references* and should be reset to $\bot$ when n is collected. This garbage collection process is formalized via the following rule, which can be applied at any time during the analysis:

$$\frac{\begin{array}{c} n \notin Range(\mathcal{C}) \qquad n \notin Range(\mathcal{H}) \\ \mathcal{L}' = \mathcal{L} \setminus \{n\} \qquad \mathcal{U}' = \mathcal{U} \setminus \{n\} \\ \mathcal{R}' = \mathcal{R} \setminus \{n\} \qquad \mathcal{W}' = \mathcal{W} \setminus \{n\} \qquad \mathcal{H}' = \mathcal{H} \setminus \{n\} \end{array}}{(\mathcal{C}, \mathcal{L}, \mathcal{U}, \mathcal{R}, \mathcal{W}, \mathcal{H}) \Rightarrow^{gc} (\mathcal{C}, \mathcal{L}', \mathcal{U}', \mathcal{R}', \mathcal{W}', \mathcal{H}')}$$

The rule uses the following notation to update maps and relations:

$$\mathcal{W} \setminus \{n\} \stackrel{\text{def}}{=} \lambda x.\ \text{if } \mathcal{W}(x) = n \text{ then } \bot \text{ else } \mathcal{W}(x)$$
$$\mathcal{H} \setminus \{n\} \stackrel{\text{def}}{=} \{(n_1, n_2) \in \mathcal{H} \mid n_1 \neq n,\ n_2 \neq n\}$$

In practice, we trigger garbage collection by including in each node a count of the number of references to that node from within $\mathcal{H}$ or $\mathcal{C}$. We maintain the invariant that the happens-before graph is acyclic, since any attempt to add a cycle-generating edge indicates an error that is immediately reported. Thus, the absence of cycles means that reference counting immediately collects all nodes as soon as they become garbage.

The experimental results of Section 6 show that garbage collection is extremely effective; we typically have at most a few dozen live nodes at any time, even for sizeable benchmarks.

4.2 Non-Transactional Operations

The rule [INS OUTSIDE] described above is inefficient, in that it allocates nodes at an extremely fast rate (one node per non-transactional heap access) and leads to long sequences of unary transactions.

In many situations, this allocation is unnecessary. In particular, for an operation $rd(t, x, v)$ outside a transaction, the [INS OUTSIDE] rule creates a new node n with predecessors $\mathcal{L}(t)$ and $\mathcal{W}(x)$. Suppose, however, that $\mathcal{L}(t)$ and $\mathcal{W}(x)$ are already $\bot$, perhaps because they have already been collected. In this case, n would be

Figure 3: Optimized Rules for Non-Transactional Operations

[INS OUTSIDE ACQUIRE]

$$\frac{\mathcal{C}(t) = \bot \quad \langle \mathcal{H}',\, n\rangle \;=\; merge(\mathcal{H}, \{\mathcal{L}(t), \mathcal{U}(m)\}) \quad \mathcal{L}' \;=\; \mathcal{L}[t := n]}{(\mathcal{C},\mathcal{L},\mathcal{U},\mathcal{R},\mathcal{W},\mathcal{H}) \Rightarrow^{acq(t,m)} (\mathcal{C},\mathcal{L}',\mathcal{U},\mathcal{R},\mathcal{W},\mathcal{H}')}$$

[INS OUTSIDE RELEASE]

$$\frac{\mathcal{C}(t) = \bot \quad \mathcal{U}' = \mathcal{U}[m := \mathcal{L}(t)]}{(\mathcal{C},\mathcal{L},\mathcal{U},\mathcal{R},\mathcal{W},\mathcal{H}) \Rightarrow^{rel(t,m)} (\mathcal{C},\mathcal{L},\mathcal{U}',\mathcal{R},\mathcal{W},\mathcal{H})}$$

[INS OUTSIDE READ]

$$\frac{\mathcal{C}(t) = \bot \quad \langle \mathcal{H}',\, n\rangle = merge(\mathcal{H}, \{\mathcal{L}(t), \mathcal{W}(x)\}) \quad \mathcal{R}' = \mathcal{R}[t := n] \qquad \mathcal{L}' = \mathcal{L}[t := n]}{(\mathcal{C},\mathcal{L},\mathcal{U},\mathcal{R},\mathcal{W},\mathcal{H}) \Rightarrow^{rd(t,x,v)} (\mathcal{C},\mathcal{L}',\mathcal{U},\mathcal{R}',\mathcal{W},\mathcal{H}')}$$

[INS OUTSIDE WRITE]

$$\frac{\mathcal{C}(t) = \bot \quad S \;=\; \{\mathcal{R}(x,t') \mid t' \in Tid\} \cup \{\mathcal{W}(x), \mathcal{L}(t)\} \quad \langle \mathcal{H}',\, n\rangle \;=\; merge(\mathcal{H}, S) \quad \mathcal{W}' = \mathcal{W}[t := n] \qquad \mathcal{L}' = \mathcal{L}[t := n]}{(\mathcal{C},\mathcal{L},\mathcal{U},\mathcal{R},\mathcal{W},\mathcal{H}) \Rightarrow^{wr(t,x,v)} (\mathcal{C},\mathcal{L}',\mathcal{U},\mathcal{R}',\mathcal{W},\mathcal{H}')}$$

$$merge \quad : \quad ((Node \times Node) \times 2^{Node_\bot}) \to ((Node \times Node) \times Node_\bot)$$

$$merge(\mathcal{H}, \{n_1, \ldots, n_k\}) \;=\; \begin{cases} \langle \mathcal{H},\, \bot\rangle & \text{if } n_i = \bot \;\forall i \in 1..k \\ \langle \mathcal{H},\, n_j\rangle & \text{if } \exists j \text{ such that } n_j \neq \bot \text{ and } \forall i \in 1..k,\; n_i = \bot \text{ or } (n_i, n_j) \in \mathcal{H}^* \\ \langle \mathcal{H} \uplus \{(n_i, n) \mid i \in 1..k\},\, n\rangle & \text{otherwise, where } n \text{ is fresh} \end{cases}$$

immediately collected once the operation finishes, and so we avoid allocating it in the first place, via the rule:

$$\frac{\mathcal{C}(t) = \bot \quad \mathcal{L}(t) = \bot \quad \mathcal{W}(x) = \bot \quad \mathcal{R}' = \mathcal{R}[(x,t) := \bot]}{(\mathcal{C},\mathcal{L},\mathcal{U},\mathcal{R},\mathcal{W},\mathcal{H}) \Rightarrow^{rd(t,x,v)} (\mathcal{C},\mathcal{L},\mathcal{U},\mathcal{R}',\mathcal{W},\mathcal{H})}$$

Alternatively, if $\mathcal{W}(x)$ is $\bot$ but $\mathcal{L}(t)$ is not, then $\mathcal{L}(t)$ is the unique predecessor of n, and n will never have additional incoming edges. Hence, the nodes $\mathcal{L}(t)$ and n can be *merged* without introducing additional cycles in the happens-before graph, or in later versions of that graph. This reasoning is summarized by the following rule:

$$\frac{\mathcal{C}(t) = \bot \quad \mathcal{L}(t) \neq \bot \quad \mathcal{W}(x) = \bot \quad \mathcal{R}' = \mathcal{R}[(x,t) := \mathcal{L}(t)]}{(\mathcal{C},\mathcal{L},\mathcal{U},\mathcal{R},\mathcal{W},\mathcal{H}) \Rightarrow^{rd(t,x,v)} (\mathcal{C},\mathcal{L},\mathcal{U},\mathcal{R}',\mathcal{W},\mathcal{H})}$$

Even if neither $\mathcal{L}(t)$ nor $\mathcal{W}(x)$ are $\bot$, we we can still re-use $\mathcal{L}(x)$ if there is a happens-before path from $\mathcal{W}(x)$ to $\mathcal{L}(x)$:

$$\frac{\mathcal{C}(t) = \bot \quad \mathcal{L}(t) \neq \bot \quad \mathcal{W}(x) \neq \bot \quad (\mathcal{L}(t), \mathcal{W}(x)) \in \mathcal{H}^* \quad \mathcal{R}' = \mathcal{R}[(x,t) := \mathcal{L}(t)]}{(\mathcal{C},\mathcal{L},\mathcal{U},\mathcal{R},\mathcal{W},\mathcal{H}) \Rightarrow^{rd(t,x,v)} (\mathcal{C},\mathcal{L},\mathcal{U},\mathcal{R}',\mathcal{W},\mathcal{H})}$$

To avoid a multitude of such rules, we introduce the auxiliary function *merge* to identify various situations where merging can safely be performed: see Figure 3. This function takes as input a happens-before relation and a collection of argument nodes $n_1, \ldots, n_k$, and it returns a (possibly extended) happens-before relation, and a (possibly new) node that happens-after each of the argument nodes. The figure also includes analysis rules that leverage *merge* to handle non-transactional operations efficiently along the lines outlined above.

4.3 Blame Assignment

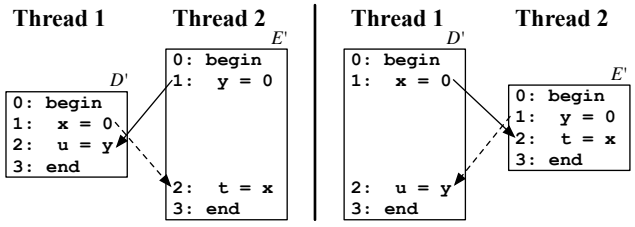
When the analysis determines that a particular trace α is not serializable, it can produce a cycle of transactions whose combination is not serializable. We now investigate how to assign blame to a particular transaction in that cycle.

A transaction A *is self-serializable in trace* α if α has an equivalent trace α' in which A executes serially. (Other transactions in α' need not execute serially, and so the notions of self-serializable transactions and serializable traces are distinct.) Once our algorithm identifies a non-serializable trace α, we would like to assign blame to a particular transaction within that trace that is not self-serializable. For the cycle $A \lessdot B' \lessdot C' \lessdot A$ described in the introduction, we should assign blame to transaction A, since all other transactions in the cycle are self-serializable.

To support blame assignment, we extend the happens-before graph to identify the particular operations inducing each edge between transactions. Specifically, we store with each edge the timestamp of the operations at its head and tail. We assign blame using these timestamps as follows. First, note that when an operation d performed by thread t during some transaction D completes the first cycle in the happens-before graph, the trace prior to d is serializable. Thus transactions other than D are still serializable, and we can only potentially blame D. From the transactional happens-before cycle, we know that $D \lessdot_\alpha E \lessdot_\alpha D$, where E is some transaction of another thread. Hence, there exists some earlier operations $d' \in D$ and $e \in E$ such that $d' \ll\!\sim_\alpha e \ll\!\sim_\alpha d$. The key question is whether $d' <_\alpha e <_\alpha d$; if so, then transaction D is not self-serializable and should be blamed.

Let n be the node for D. For each node $m \neq n$ on the happens-before cycle, if the timestamp on the incoming edge to m is less than or equal to the timestamp on the outgoing edge from m, then the cycle is said to be *increasing*. In this situation, the happens-before relation on transactions reflects the underlying happens-before relation on operations, and so there do exist some earlier operations $d' \in D$ and $e \in E$ such that $d' <_\alpha e <_\alpha d$. Hence the transaction D, which contains both d' and d, is not self-serializable.

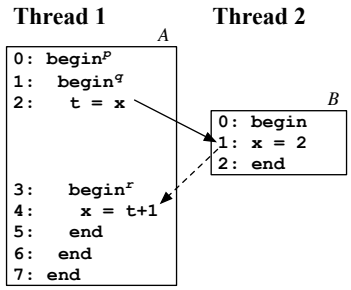
Somewhat surprisingly, it is not always possible to blame a single transaction, since all transactions in a non-serializable trace may still be self-serializable. To illustrate this point, consider the following two traces. The first trace executes D' serially, but transaction E' is also self-serializable (as illustrated by the second, equivalent trace). Thus, both transactions are self-serializable, even though together they constitute a non-serializable trace.



Despite this theoretical difficulty, in practice our algorithm is generally successful at assigning blame for each non-serializable trace to a particular transaction that is not self-serializable.

Nested Atomic Blocks. We now extend our system to nested atomic blocks. Only the outermost atomic block is considered to start a new transaction; nested blocks execute within that transaction but can still be refuted by our blame assignment algorithm.

To support nesting, we extend $\mathcal{C}(t)$ to denote a stack, where the entries record both the identifying label and the timestamp of the first operation in each atomic block in the dynamic scope. For example, right before Thread 1 executes step 4 below, $\mathcal{C}(1)$ would contain $(p, 0).(q, 1).(r, 3)$:



Once Thread 1 executes step 4, our algorithm detects an increasing cycle from A and will refute any atomic block in A that contains both the root and target operations ("`2: t = x`" and "`4: x = t+1`", respectively) of that cycle. Thus, the algorithm will refute the atomic blocks p and q; the block labeled r is not refuted, and is serializable (indeed, serial in this trace).

Blame Assignment Details. To implement blame assignment, we introduce the notion of a $Step$, which is a pair of a transaction node and a timestamp. We extend the component $\mathcal{U}$ of the analysis state so that $\mathcal{U}(m)$ is now a $Step$ that records both the transaction and the timestamp of the last release operation on m; the other state components are extended in a similar fashion:

$$\begin{array}{rcll} \phi & : & (\mathcal{C}, \mathcal{L}, \mathcal{U}, \mathcal{R}, \mathcal{W}, \mathcal{H}) & \\ \mathcal{C} & : & Tid & \rightarrow \quad (Label \times Step)^* \\ \mathcal{L} & : & Tid & \rightarrow \quad Step_\perp \\ \mathcal{U} & : & Lock & \rightarrow \quad Step_\perp \\ \mathcal{R} & : & Var \times Tid & \rightarrow \quad Step_\perp \\ \mathcal{W} & : & Var & \rightarrow \quad Step_\perp \\ \mathcal{H} & \subseteq & Step \times Step & \\ Step & = & Node \times Nat & \\ \phi_0 & = & (\lambda t.\epsilon,\ \lambda t.\perp,\ \lambda m.\perp,\ \lambda x, t.\perp,\ \lambda x.\perp,\ \emptyset) & \end{array}$$

The revised analysis is defined by the rules in Figure 4. The rule [INS2 ENTER] for $begin^l(t)$ handles the case where a new transaction is required because the stack $\mathcal{C}(t)$ is empty. It allocates a fresh node n, creates a step that pairs n with the initial timestamp 0 of the $begin^l(t)$ operation, and records that thread t is now executing an atomic block labeled l, where s is the first step of that block.

We disallow multiple edges with different timestamps between the same nodes in the happens-before graph, for space reasons. That is, if $((n, i), (m, j))$ and $((n, i'), (m, j'))$ are both in $\mathcal{H}$ then $i = i'$ and $j = j'$. This invariant bounds the size of $\mathcal{H}$ by $|Node|^2$. To preserve this invariant, rule [INS2 ENTER] uses the following operation to extend the happens-before relation $\mathcal{H}$ with additional edges $G \subseteq (Step_\perp \times Step_\perp)$:

$$\begin{aligned} \mathcal{H} \uplus G = \{((n, i), (m, j)) \in \mathcal{H} \mid \neg\exists i', j'.\ ((n, i'), (m, j')) \in G\} \\ \cup\ \{((n, i), (m, j)) \in G \mid n \neq m\} \end{aligned}$$

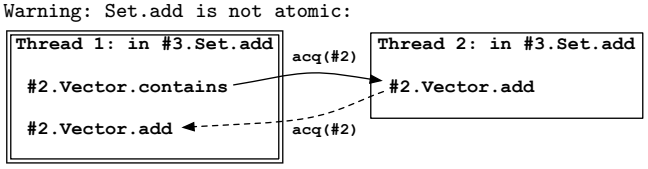
If thread t is already inside a transaction when it executes $begin^l(t)$, rule [INS2 RE-ENTER] extends the stack $\mathcal{C}(t)$ with an additional entry for the new atomic block. These rules use the notation $\mathcal{L}(t) + 1$ to increment the timestamp within a step; if $\mathcal{L}(t) = (n, k)$, then $\mathcal{L}(t) + 1 = (n, k + 1)$. The rule [INS2 EXIT] exits an atomic block by popping the last entry of the stack and, as in the other rules, incrementing the timestamp in $\mathcal{L}(t)$.

The four [INS2 OUTSIDE ...] rules are variants of the earlier [INS OUTSIDE ...] rules that use the revised $merge$ function shown in Figure 4, which operates on steps. These rules mostly ignore timestamps, since they operate on unary transactions that are by definition serializable. In particular, the $merge$ function ignores timestamps when searching for a representative step s_j that (non-strictly) happens after steps $s_1, ..., s_k$.

The garbage collection rule [INS2 GC] picks a node n such that no step in $S = \{n\} \times Nat$ occurs on any transaction stack or in the range of the happens-before relation $\mathcal{H}$; all references to steps in S are then removed from the analysis store and replaced with $\perp$.

5. Velodrome Prototype

We have developed a prototype implementation, called Velodrome, of our atomicity analysis. This tool takes as input a compiled Java program and a specification of which methods in that program should be atomic, and it reports an error whenever it observes a non-serializable trace of an atomic method. For example, on the `Set` example from the introduction, Velodrome reports the following error graph (generated with `dot` [16]) when two threads concurrently add elements to the same set object:



In this error message, `#3` denotes a particular `Set` object and `#2` denotes the set's underlying `Vector`. The boxes indicate the transactions whose combination is not serializable. Each happens-before edge is labeled with the operation that generated it, and the last edge in a cycle is dashed. The outlined box indicates where Velodrome has placed blame. These graphs are extremely useful for understanding error messages, and Velodrome records additional diagnostic information to construct them.

Velodrome is a component of RoadRunner, a general framework for implementing dynamic concurrent program analyses. RoadRunner is written entirely in Java and can run on any standard JVM. RoadRunner instruments class files at load time using the BCEL Bytecode Engineering Library [3]. The instrumented code generates an event stream, with one event for each lock acquire or release, memory read or write, and atomic method entry or exit performed by the target program. RoadRunner passes this event stream to the analysis back-end.

Working exclusively at the bytecode level offers several advantages. Specifically, the tool can check any Java program, regardless of whether the full source code is available, and only needs to reason about the relatively simple bytecode language. However, this does make it difficult to support source-level annotations, which are useful for specifying which methods should be atomic, specifying library behaviors, and so on. We currently support such configuration through command-line options.

Re-entrant (and hence redundant) lock acquires and releases are filtered out by RoadRunner, and so do not complicate the back-end analysis. RoadRunner is typically configured to also filter out operations on thread-local data, which dramatically improves the performance of the analyses, although this optimization is slightly unsound [36]. One limitation of our current prototype is that it performs the analysis only on objects and fields, and not on arrays. Supporting arrays would be possible, but would add additional complexity.

Figure 4: Instrumentation Relation with Blame Assignment

[INS2 ENTER]
$$\frac{\mathcal{C}(t) = \epsilon \quad s = (n,0) \text{ where } n \text{ is fresh} \quad \mathcal{C}' = \mathcal{C}[t := (l,s)] \quad \mathcal{L}' = \mathcal{L}[t := s] \quad \mathcal{H}' = \mathcal{H} \uplus \{(\mathcal{L}(t), s)\}}{(\mathcal{C},\mathcal{L},\mathcal{U},\mathcal{R},\mathcal{W},\mathcal{H}) \Rightarrow^{begin^l(t)} (\mathcal{C}',\mathcal{L}',\mathcal{U},\mathcal{R},\mathcal{W},\mathcal{H}')}$$

[INS2 RE-ENTER]
$$\frac{\mathcal{C}(t) = \beta \neq \epsilon \quad s = \mathcal{L}(t)+1 \quad \mathcal{C}' = \mathcal{C}[t := \beta.(l,s)] \quad \mathcal{L}' = \mathcal{L}[t := s] \quad \mathcal{H}' = \mathcal{H} \uplus \{(\mathcal{L}(t), s)\}}{(\mathcal{C},\mathcal{L},\mathcal{U},\mathcal{R},\mathcal{W},\mathcal{H}) \Rightarrow^{begin^l(t)} (\mathcal{C}',\mathcal{L}',\mathcal{U},\mathcal{R},\mathcal{W},\mathcal{H}')}$$

[INS2 EXIT]
$$\frac{\mathcal{C}(t) = \beta.(s',l) \quad s = \mathcal{L}(t)+1 \quad \mathcal{C}' = \mathcal{C}[t := \beta] \quad \mathcal{L}' = \mathcal{L}[t := s]}{(\mathcal{C},\mathcal{L},\mathcal{U},\mathcal{R},\mathcal{W},\mathcal{H}) \Rightarrow^{end(t)} (\mathcal{C}',\mathcal{L}',\mathcal{U},\mathcal{R},\mathcal{W},\mathcal{H})}$$

[INS2 GC]
$$\frac{S = \{n\} \times Nat \quad \forall t.\ (\mathcal{C}(t) \neq \epsilon \ \Rightarrow\ \mathcal{L}(t) \notin S) \quad \mathcal{H} \cap (Step \times S) = \emptyset \quad \mathcal{H}' = \mathcal{H} \setminus S \quad \mathcal{L}' = \mathcal{L} \setminus S \quad \mathcal{R}' = \mathcal{R} \setminus S \quad \mathcal{W}' = \mathcal{W} \setminus S \quad \mathcal{U}' = \mathcal{U} \setminus S}{(\mathcal{C},\mathcal{L},\mathcal{U},\mathcal{R},\mathcal{W},\mathcal{H}) \Rightarrow^{gc} (\mathcal{C},\mathcal{L}',\mathcal{U}',\mathcal{R}',\mathcal{W}',\mathcal{H}')}$$

[INS2 INSIDE ACQUIRE]
$$\frac{\mathcal{C}(t) \neq \epsilon \quad s = \mathcal{L}(t)+1 \quad \mathcal{L}' = \mathcal{L}[t := s] \quad \mathcal{H}' = \mathcal{H} \uplus \{(\mathcal{U}(m), s)\}}{(\mathcal{C},\mathcal{L},\mathcal{U},\mathcal{R},\mathcal{W},\mathcal{H}) \Rightarrow^{acq(t,m)} (\mathcal{C},\mathcal{L}',\mathcal{U},\mathcal{R},\mathcal{W},\mathcal{H}')}$$

[INS2 INSIDE RELEASE]
$$\frac{\mathcal{C}(t) \neq \epsilon \quad s = \mathcal{L}(t)+1 \quad \mathcal{L}' = \mathcal{L}[t := s] \quad \mathcal{U}' = \mathcal{U}[m := s]}{(\mathcal{C},\mathcal{L},\mathcal{U},\mathcal{R},\mathcal{W},\mathcal{H}) \Rightarrow^{rel(t,m)} (\mathcal{C},\mathcal{L}',\mathcal{U}',\mathcal{R},\mathcal{W},\mathcal{H})}$$

[INS2 INSIDE READ]
$$\frac{\mathcal{C}(t) \neq \epsilon \quad s = \mathcal{L}(t)+1 \quad \mathcal{L}' = \mathcal{L}[t := s] \quad \mathcal{R}' = \mathcal{R}[(x,t) := s] \quad \mathcal{H}' = \mathcal{H} \uplus \{(\mathcal{W}(x), s)\}}{(\mathcal{C},\mathcal{L},\mathcal{U},\mathcal{R},\mathcal{W},\mathcal{H}) \Rightarrow^{rd(t,x,v)} (\mathcal{C},\mathcal{L}',\mathcal{U},\mathcal{R}',\mathcal{W},\mathcal{H}')}$$

[INS2 INSIDE WRITE]
$$\frac{\mathcal{C}(t) \neq \epsilon \quad s = \mathcal{L}(t)+1 \quad \mathcal{L}' = \mathcal{L}[t := s] \quad \mathcal{W}' = \mathcal{W}[x := s] \quad \mathcal{H}' = \mathcal{H} \uplus (\{(\mathcal{R}(x,t'), s) \mid t' \in Tid\} \cup \{(\mathcal{W}(x), s)\})}{(\mathcal{C},\mathcal{L},\mathcal{U},\mathcal{R},\mathcal{W},\mathcal{H}) \Rightarrow^{wr(t,x,v)} (\mathcal{C},\mathcal{L}',\mathcal{U},\mathcal{R},\mathcal{W}',\mathcal{H}')}$$

[INS2 OUTSIDE ACQUIRE]
$$\frac{\mathcal{C}(t) = \epsilon \quad \langle \mathcal{H}',\ s\rangle = merge(\mathcal{H}, \{\mathcal{L}(t), \mathcal{U}(m)\}) \quad \mathcal{L}' = \mathcal{L}[t := s]}{(\mathcal{C},\mathcal{L},\mathcal{U},\mathcal{R},\mathcal{W},\mathcal{H}) \Rightarrow^{acq(t,m)} (\mathcal{C},\mathcal{L}',\mathcal{U},\mathcal{R},\mathcal{W},\mathcal{H}')}$$

[INS2 OUTSIDE RELEASE]
$$\frac{\mathcal{C}(t) = \epsilon \quad s = \mathcal{L}(t)+1 \quad \mathcal{L}' = \mathcal{L}[t := s] \quad \mathcal{U}' = \mathcal{U}[m := s]}{(\mathcal{C},\mathcal{L},\mathcal{U},\mathcal{R},\mathcal{W},\mathcal{H}) \Rightarrow^{rel(t,m)} (\mathcal{C},\mathcal{L}',\mathcal{U}',\mathcal{R},\mathcal{W},\mathcal{H})}$$

[INS2 OUTSIDE READ]
$$\frac{\mathcal{C}(t) = \epsilon \quad \langle \mathcal{H}',\ s\rangle = merge(\mathcal{H}, \{\mathcal{L}(t), \mathcal{W}(x)\}) \quad \mathcal{R}' = \mathcal{R}[t := s] \quad \mathcal{L}' = \mathcal{L}[t := s]}{(\mathcal{C},\mathcal{L},\mathcal{U},\mathcal{R},\mathcal{W},\mathcal{H}) \Rightarrow^{rd(t,x,v)} (\mathcal{C},\mathcal{L}',\mathcal{U},\mathcal{R}',\mathcal{W},\mathcal{H}')}$$

[INS2 OUTSIDE WRITE]
$$\frac{\mathcal{C}(t) = \epsilon \quad S = \{\mathcal{R}(x,t') \mid t' \in Tid\} \cup \{\mathcal{W}(x), \mathcal{L}(t)\} \quad \langle \mathcal{H}',\ s\rangle = merge(\mathcal{H}, S) \quad \mathcal{W}' = \mathcal{W}[t := s] \quad \mathcal{L}' = \mathcal{L}[t := s]}{(\mathcal{C},\mathcal{L},\mathcal{U},\mathcal{R},\mathcal{W},\mathcal{H}) \Rightarrow^{wr(t,x,v)} (\mathcal{C},\mathcal{L}',\mathcal{U},\mathcal{R}',\mathcal{W},\mathcal{H}')}$$

$$merge \quad : \quad ((Step \times Step) \times 2^{Step_\perp}) \to ((Step \times Step) \times Step_\perp)$$

$$merge(\mathcal{H}, \{s_1, \ldots, s_k\}) \quad = \quad \begin{cases} \langle \mathcal{H},\ \perp\rangle & \text{if } s_i = \perp\ \forall i \in 1..k \\ \langle \mathcal{H},\ s_j\rangle & \text{if } \exists j \text{ such that } s_j \neq \perp \text{ and} \\ & \quad \forall i \in 1..k,\ s_i = \perp \text{ or } s_i \text{ happens-before } s_j \text{ in } \mathcal{H} \\ \langle \mathcal{H} \uplus \{(s_i, s) \mid i \in 1..k\},\ s\rangle & \text{otherwise, where } s = (n,0) \text{ and } n \text{ is fresh} \end{cases}$$

RoadRunner includes several race detection algorithms (including Eraser [36] and a complete happens-before detector), which can be run concurrently with Velodrome if race conditions are a concern in the target program. As mentioned in the introduction, Velodrome and the Atomizer can also be run concurrently.

Analysis Store. Efficiently implementing the analysis of Figure 4 requires a number of careful data representation choices, particularly for handling weak references to nodes that are internally collected and recycled by our analysis. Each step is represented as a 64-bit integer whose top 16 bits identify a particular `Node` object, and whose lower 48 bits represent a timestamp within that `Node`. When a `Node` n is collected, we also record the last timestamp k used for that `Node`. If a step (n, k') is later dereferenced, we check whether $k' \leq k$; if so, then that step is interpreted as being $\perp$, since the conceptual node it pointed to has been collected, even though the corresponding `Node` object has been recycled to represent a new conceptual node.

For each node, we maintain a set of ancestors of that node. This ancestor set allows us to immediately detect when a cycle is about to be added to the graph, which yields several benefits: (1) It supports more precise error messages, which could include, for example, the stack trace of the current thread. (2) It enables us to avoid adding that edge and thus maintain an acyclic graph, which facilitates reference-counting garbage collection. (3) It supports an efficient implementation of the $merge$ function of Figure 4.

Adversarial Scheduling. As mentioned in the introduction, our system can guide the scheduler to generate traces likely to exhibit atomicity violations. In particular, we can configure Velodrome to concurrently perform the Atomizer analysis and temporarily suspend any thread that is about to perform an operation leading to a potential atomicity violation. This delay, which we currently set to 100 milliseconds, increases the probability that other threads will perform conflicting operations and yield a non-serializable trace that is then caught by Velodrome. We are exploring a number of other scheduling policies, such as pausing writes but not reads,

Program	Size (lines)	Base Time (sec.)	Instrumented Time (slowdown)				Velodrome Transactions			
							Without Merge		With Merge	
			Empty	Eraser	Atomizer	Velodrome	Allocated	Max. Alive	Allocated	Max. Alive
elevator	520	5.64	1.1	1.1	1.1	1.1	174,000	20	170,000	13
hedc	6,400	0.21	6.2	6.0	5.9	6.3	79	37	58	4
tsp	700	0.46	30.9	50.9	60.2	71.7	>1,000,000	8	12,000	1
sor	690	0.34	2.3	2.3	2.4	2.9	2,000	2	2	2
jbb	36,000	9.84	2.9	3.2	3.4	3.1	21,000	9	14,000	13
mtrt	11,000	0.85	9.3	14.3	22.4	18.3	645,000	5	645,000	5
moldyn	1,400	0.77	3.8	4.0	4.1	4.5	5	4	5	4
montecarlo	3,600	1.70	1.6	1.7	1.7	1.7	410,000	4	300,000	4
raytracer	18,000	2.00	4.5	6.7	9.4	9.2	128	8	23	8
colt	29,000	16.40	1.2	1.2	1.2	1.2	113	11	58	19
philo	84	2.71	1.0	1.0	1.2	1.2	34	5	34	5
raja	10,000	0.55	4.3	4.4	4.5	4.5	60	1	60	1
multiset	300	0.10	4.0	4.4	4.7	10.0	218,000	8	8	8
webl	22,300	0.52	8.6	8.9	9.3	21.0	470,000	4	395,000	4
jigsaw	91,100	8.2	1.1	1.1	1.1	1.1	123,000	99	36,600	17

Table 1: Benchmark sizes and running times, analysis slowdowns, and happens-before graph statistics.

allowing some threads to never pause, and so on. Similar techniques have proven quite effective in other contexts [20].

6. Evaluation

Benchmarks This section summarizes our experience applying Velodrome to several benchmark programs: `elevator`, a discrete event simulator for elevators [41]; `hedc`, a tool to access astrophysics data from Web sources [41]; `tsp`, a Traveling Salesman Problem solver [41]; `sor`, a scientific computing program [41]; `mtrt`, a multithreaded ray-tracing program from the SPEC JVM98 benchmark suite [39]; `specJBB`, the SPEC JBB2000 business object simulator [39]; `moldyn`, `montecarlo`, and `raytracer` from the Java Grande benchmark suite [24]; the `colt` scientific computing library [5]; the `raja` ray tracer [15]; and `multiset`, a basic multiset implementation, `philo`, a dining philosophers simulation [7]; `webl`, a scripting language interpreter for processing web pages, configured to execute a simple web crawler [25]; and `jigsaw`, an open source web server [45] configured to serve a fixed number of pages to a crawler.

We performed all experiments on an Apple Mac Pro with a quad-core 3GHz Pentium Xeon processor and 2GB of memory, using OS X 10.4 and Sun's Java HotSpot Client VM, version 1.5.0. All classes loaded by the benchmark programs were instrumented, except those from the standard Java libraries.

Table 1 presents the size and uninstrumented running time of each program, as well as the slowdown of each program when instrumented and analyzed by four back-end analyses: Empty (which does no work and simply measures the instrumentation overhead), Eraser [36], Atomizer [11], and Velodrome (using the optimized semantics of Figure 4).

To make the performance experiments realistic, we used Velodrome to identify non-atomic methods and configured the Atomizer and Velodrome to only check the remaining methods for atomicity. This configuration mimics using these tools in contexts where most or all methods satisfy their atomicity specification. This configuration actually increases the overhead for Velodrome when compared to checking *all* methods for atomicity, because program traces contain many small transactions rather than a few monolithic ones.

Overall, the performance of Velodrome is quite promising, and mostly competitive with the less precise Eraser and Atomizer algorithms. (It should be noted that `elevator`, `hedc`, `philo`, `webl`, and `jigsaw` are not compute-bound, but the remaining benchmarks are and experienced an average slowdown of 9.3 for Eraser, 10.4 for Atomizer, and 12.7 for Velodrome.) We believe the substantial slowdown for `tsp` is due to the instrumented code preventing the virtual machine's adaptive compiler from performing certain optimizations.

The last four columns of Table 1 highlight the impact of node merging and garbage collection from Section 4. The columns labeled "Allocated" and "Max. Alive" under the heading "Without Merge" show the total number of graph nodes allocated and the maximum number that are active at any time during execution when using the naïve [INS OUTSIDE] rule. The columns under "With Merge" show the same measurements for the final, optimized semantics of Figure 4. Two important observations are worth noting: (1) Garbage collection is essential and extremely effective, reducing the number of live nodes by up to four orders of magnitude. (2) Merging reduces the number of node allocations by up to several orders of magnitude and has a dramatic impact on running times.

Table 2 presents the number of methods for which Atomizer generated warnings, under the assumption that *all* methods are atomic. For these measurements, we counted the number of distinct warnings over a series of five runs. We classified each Atomizer warning either as actually corresponding to a non-atomic method (that is, a method that is not serializable in some trace) or as a false alarm. The non-atomic methods include several errors reported in earlier work [11, 32], as well as methods that were not intended to be atomic (such as Thread `run()` methods and similar routines). The false alarms were due to imprecision in the Atomizer's underlying race condition and reduction analyses and its inability to reason about non-lock-based synchronization. That table also shows the number of non-atomic methods found by Velodrome during the five runs, as well as how many non-atomic methods reported by the Atomizer were missed by Velodrome. For both tools, the large majority of errors were reported on the first of the five runs.

Overall, these results indicate that Velodrome is quite effective at identifying non-atomic methods. As expected, Velodrome's completeness (and hence lack of generalization) did occasionally cause it to require more runs to find some errors, and it did miss a small number of non-atomic methods in some programs because the execution of these methods happened to be serializable in the observed traces. In `jigsaw`, 6 of the missed warnings were due to a single non-atomic method that Velodrome mischaracterized. On most benchmarks, however, Velodrome did identify most or all non-atomic methods identified by the Atomizer.

Moreover, Velodrome did not report any of the large number of false alarms generated by the Atomizer. In particular, it avoided reporting many spurious warnings on `jbb` and `mtrt` caused by imprecise race analysis, `fork-join` synchronization, and other idioms

Program	Warnings				
	Atomizer		Velodrome		
	Non-Serial	False Alarms	Non-Serial	False Alarms	Missed
elevator	5	1	5	0	0
hedc	6	2	6	0	0
tsp	8	0	8	0	0
sor	3	0	3	0	0
jbb	5	42	5	0	0
mtrt	2	27	2	0	0
moldyn	4	0	4	0	0
montecarlo	6	0	6	0	0
raytracer	2	3	1	0	1
colt	27	2	20	0	7
philo	2	0	2	0	0
raja	0	0	0	0	0
multiset	5	0	5	0	0
webl	24	2	22	0	2
jigsaw	55	5	44	0	11
Total	**154**	**84**	**133**	**0**	**21**

Table 2: Warnings produced by the Atomizer and Velodrome, under the assumption that all methods should be atomic.

not understood by the Atomizer. The `mtrt` code also makes heavy use of the standard Java libraries, which are not instrumented. As such, Atomizer cannot reason about synchronization performed inside those libraries and generates many warnings as a result. Velodrome does not suffer from this limitation: if Velodrome observes a subsequence α' of the actual program trace α, then if α' is not serializable it follows that α is also not serializable. Thus, uninstrumented libraries do not cause Velodrome to report false alarms.

Interestingly, the number of warnings produced was fairly uniform when these experiments were repeated using only a single core, despite Velodrome being more sensitive to scheduling than other tools. This may not always be the case but additional experimentation on large programs is needed to fully quantify the impact of the number of cores on Velodrome's analysis. Also, Velodrome's blame assignment algorithm is quite effective, and assigned blame to a specific method for over 80% of the warnings.

In summary, Velodrome dramatically reduces the false alarm rate in comparison to the Atomizer. Roughly half of the Atomizer warnings are false alarms, but Velodrome produces none, while still detecting almost all (85%) of the non-atomic methods.

Using the Atomizer to adjust the scheduler, as described in the previous section, improved Velodrome's ability to find defects during several small experiments. Velodrome found the second non-serial method in `raytracer`, as well as one additional non-serial method in `colt` and several more in `jigsaw`. To further study this technique, we injected atomicity defects into two programs, `elevator` and `colt`, by systematically removing each synchronized statement that induced contention between threads one at a time and then running our analysis on each corrupted program. Without scheduler adjustments, a single run by Velodrome found the inserted defect approximately 30% of the time. With scheduler adjustments, the success rate increased to approximately 70%.

7. Related Work

A variety of tools have been developed to check for atomicity violations, both statically and dynamically. The Atomizer [11] uses Lipton's theory of reduction [27] to check whether steps of each transaction conform to a pattern guaranteed to be serializable.

Wang and Stoller developed an alternative *block-based* approach to verifying atomicity. This approach is more precise than reduction-based approaches, but it is significantly slower for some programs. A detailed experimental comparison of the two approaches is presented in [44]. Wang and Stoller also developed more precise *commit-node* algorithms [43]. These algorithms focus on both *conflict-atomicity* (referred to simply as *atomicity* in this paper) and *view-atomicity*. By design, these algorithms detect serializability violations that do not occur on the current interleaving but which could occur on other interleavings. Of course, those other interleavings may not be feasible under the program's semantics, so these algorithms may yield false alarms.

In other work, Xu, Bodik, and Hill [46] developed a precise dynamic analysis for enforcing *Strict 2-Phase Locking* [9], a sufficient but not necessary condition for ensuring serializability. Hence violations, while possibly worthy of investigation, do not necessarily imply that the observed trace is not serializable. An alternative approach for verifying atomicity using model-checking has been explored by Hatcliff *et al*. [19]. Their results suggest that checking atomicity with model-checking is feasible for unit-testing, where the reachable state space is relatively small.

Other work explores static analyses, including approaches to statically compute and look for cycles in the happens-before graph [10]. Type systems [14, 2, 12, 35, 13, 42] have also been investigated. Compared to dynamic techniques, static systems provide stronger soundness guarantees and detect errors earlier in the development cycle, but many of them require more effort from the programmer or are limited in precision and scalability. To date, none can yet handle all synchronization disciplines precisely.

Hoare [22] and Lomet [29] first proposed the use of atomic blocks for synchronization, and the Argus [28] and Avalon [8] projects developed language support for implementing atomic objects. More recent studies have focused on lightweight transactions [38, 18, 23] and automatic generation of synchronization code from high-level specifications [6, 31, 40, 21]. Much of this work is orthogonal to ours, and while these approaches offer a promising alternative concurrency control, we believe that a combination of the two approaches will be the most effective programming model for the foreseeable future.

8. Conclusions

Programmer support for building reliable multithreaded programs will only continue to grow in importance. Despite the successes of previous race condition and atomicity checkers, the need to identify false alarms places a large burden on the programmer. We have presented a sound and complete atomicity checker that finds almost all of the atomicity violations found by less precise tools, and which guarantees each warning represents a real violation of conflict-serializability. Our tool occasionally misses a warning that would be produced by other tools because it does not generalize the observed trace to reason about behavior under different schedules. To close this coverage gap, we are exploring ways to guide execution toward traces most likely to contain real atomicity errors.

Acknowledgments

This work was supported in part by the National Science Foundation under Grants 0341179, 0341387, and 0644130, and by a Sloan Foundation Fellowship. We thank Kenn Knowles and Caitlin Sadowski for formalizing parts of the Velodrome metatheory and clarifying aspects of our development. We thank Tayfun Elmas, Christoph von Praun, and Ben Wood for their assistance with the benchmark programs.

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 *International Conference on Automated Software Engineering*, pages 233–242, 2005.

[2] R. Agarwal and S. D. Stoller. Type inference for parameterized race-free Java. In *Conference on Verification, Model Checking, and Abstract Interpretation*, pages 149–160, 2004.

[3] BCEL. `http://jakarta.apache.org/bcel`, 2007.

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

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

[6] X. Deng, M. Dwyer, J. Hatcliff, and M. Mizuno. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *International Conference on Software Engineering*, pages 442–452, 2002.

[7] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware Java runtime. In *Conference on Programming Language Design and Implementation*, pages 245–255, 2007.

[8] J. L. Eppinger, L. B. Mummert, and A. Z. Spector. *Camelot and Avalon: A Distributed Transaction Facility*. 1991.

[9] K. P. Eswaran, J. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.

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

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

[12] C. Flanagan, S. N. Freund, and M. Lifshin. Type inference for atomicity. In *Workshop on Types in Language Design and Implementation*, pages 47–58, 2005.

[13] C. Flanagan, S. N. Freund, and S. Qadeer. Exploiting purity for atomicity. *IEEE Trans. Soft. Eng.*, 31(4):275–291, 2005.

[14] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Conference on Programming Language Design and Implementation*, pages 338–349, 2003.

[15] E. Fleury and G. Sutre. Raja, version 0.4.0-pre4. `http://raja.sourceforge.net/`, 2007.

[16] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software Practice Experience*, 30(11):1203–1233, 2000.

[17] K. Gharachorloo. *Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Stanford University, 1995.

[18] T. Harris and K. Fraser. Language support for lightweight transactions. In *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 388–402, 2003.

[19] J. Hatcliff, Robby, and M. B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In *International Conference on Verification, Model Checking and Abstract Interpretation*, pages 175–190, 2004.

[20] K. Havelund. Using runtime analysis to guide model checking of Java programs. In *SPIN Model Checking and Software Verification*, pages 245–264, 2000.

[21] M. Hicks, J. S. Foster, and P. Pratikakis. Inferring locking for atomic sections. In *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[22] C. A. R. Hoare. Towards a theory of parallel programming. In *Operating Systems Techniques*, volume 9 of *A.P.I.C. Studies in Data Processing*, pages 61–71, 1972.

[23] S. Jagannathan, J. Vitek, A. Welc, and A. L. Hosking. A transactional object calculus. *Sci. Comput. Program.*, 57(2):164–186, 2005.

[24] Java Grande Forum. Java Grande benchmark suite. `http://www.javagrande.org/`, 2003.

[25] T. Kistler and J. Marais. WebL – a programming language for the web. In *World Wide Web Conference*, pages 259–270, 1998.

[26] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

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

[28] B. Liskov, D. Curtis, P. Johnson, and R. Scheifler. Implementation of Argus. In *Symposium on Operating Systems Principles*, pages 111–122, 1987.

[29] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. *Language Design for Reliable Software*, pages 128–137, 1977.

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

[31] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: synchronization inference for atomic sections. In *Symposium on Principles of Programming Languages*, pages 346–358, 2006.

[32] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Symposium on Principles and Practice of Parallel Programming*, pages 167–178, 2003.

[33] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *Symposium on Principles and Practice of Parallel Programming*, pages 179–190, 2003.

[34] M. Ronsse and K. D. Bosschere. RecPlay: A fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, 1999.

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

[36] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[37] E. Schonberg. On-the-fly detection of access anomalies. In *Conference on Programming Language Design and Implementation*, pages 285–297, 1989.

[38] N. Shavit and D. Touitou. Software transactional memory. In *Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[39] Standard Performance Evaluation Corporation. SPEC benchmarks. `http://www.spec.org/`, 2003.

[40] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *Symposium on Principles of Programming Languages*, pages 334–345, 2006.

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

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

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

[44] L. Wang and S. D. Stoller. Runtime analysis of atomicity for multi-threaded programs. *IEEE Trans. Soft. Eng.*, 32:93–110, Feb. 2006.

[45] World Wide Web Consortium. Jigsaw. `http://www.w3c.org`, 2001.

[46] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *Conference on Programming Language Design and Implementation*, pages 1–14, 2005.

[47] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *Symposium on Operating System Principles*, pages 221–234, 2005.