# SingleTrack: A Dynamic Determinism Checker for Multithreaded Programs

Caitlin Sadowski[1] Stephen N. Freund[2] Cormac Flanagan[1]

[1] University of California at Santa Cruz, Santa Cruz, CA
[2] Williams College, Williamstown, MA

**Abstract.** Multithreaded programs are prone to errors caused by unintended interference between concurrent threads. This paper focuses on verifying that *deterministically-parallel* code is free of such thread interference errors. Deterministically-parallel code may create and use new threads, via fork and join, and coordinate their behavior with synchronization primitives, such as barriers and semaphores. Such code does not satisfy the traditional non-interference property of atomicity (or serializability), however, and so existing atomicity tools are inadequate for checking deterministically-parallel code. We introduce a new non-interference specification for deterministically-parallel code, and we present a dynamic analysis to enforce it. We also describe SingleTrack, a prototype implementation of this analysis. SingleTrack's performance is competitive with prior atomicity checkers, but it produces many fewer spurious warnings because it enforces a more general non-interference property that is applicable to more software.

## 1 Introduction

Multiple threads of control are widely used in software development for many reasons, including their ability to utilize modern multi-core processors. Reasoning about the correctness of multithreaded code is notoriously difficult, however, due to the potential for non-deterministic interference between threads. Thus, methods for specifying and controlling thread interference are crucial for the cost-effective development of reliable multithreaded software. Previous studies have explored analyses for controlling interference by verifying, for example, that a program is free of data races or that methods are atomic (in that they always behave as if they execute serially). Some programs, however, are safe despite the presence of non-atomic methods, and previous studies revealed numerous examples of such methods. Motivated by this experience, this paper explores a more general non-interference property, namely *deterministic parallelism.*

*Deterministic Parallelism.* A deterministically-parallel computation may use multiple threads, but these threads either do not communicate (as in divide-and-conquer parallelism) or they communicate in a deterministic manner (e.g., via barriers). In either case, the relative scheduling of threads in subcomputations does not affect the program's overall behavior.

**Figure 1: Deterministically Parallel Sort Implementation**

```
deterministic void quicksort(int[] a) {
  synchronized (a) {
    quicksort_helper(a, 0, a.length-1);
  }
}

void quicksort_helper(int[] a, int lo, int hi) {
  if (hi - lo > 1) {
    int pivot = partition(a, lo, hi);
    Thread t1 = fork { quicksort_helper(a, lo, pivot-1); }
    Thread t2 = fork { quicksort_helper(a, pivot+1, hi); }
    t1.join();
    t2.join();
  }
}
```

To illustrate this concurrency pattern, consider the multithreaded `quicksort` implementation shown in Figure 1. That method synchronizes on the lock for array `a`, and then calls a helper method to sort the array by partitioning it and forking two threads to recursively sort each half. The `quicksort` method is annotated with the non-interference specification "`deterministic`." Each invocation of `quicksort` produces a computation involving multiplie threads: the initial thread and all threads forked by the `quicksort_helper` method. We refer to the execution of a `deterministic` method and its forked threads as a *transaction.*

In general, a program execution may involve multiple, possibly concurrent, transactions, and each transaction may be internally multithreaded (if, as in the `quicksort` function, its code forks new threads). The goal of this paper is to verify that the entire program execution satisfies the following two important non-interference properties. These two properties prevent interference problems between threads in one transaction, and in different transactions, respectively.

1. **Conflict Freedom.** Threads insinde each transaction must be *conflict-free.* That is, if two operations from the same transaction are enabled at the same time, then those operations must not conflict. Thus, all *intra-transaction* race conditions are forbidden, including those on regular variables, on `volatile` variables, and on locks. Deterministic synchronization, such as fork-join patterns and barrier synchronization, is allowed, as is synchronization *between* transactions, as in the quicksort example.
2. **External Serializability.** Threads inside each transaction must not interfere with threads outside that transaction. Note that this notion is different than atomicity, which would require the `quicksort` method to behave as if it executes serially, without interleaved operations from other threads. Since `quicksort_helper` must wait for the forked threads to terminate, `quicksort` cannot execute serially and is not atomic.
   Nevertheless, `quicksort` does enjoy a strong atomicity-like property, but only when considering the operations of all threads in the entire `quicksort` transaction, and not just the operations of the thread calling `quicksort`.

More specifically, a trace is *externally serial* if each (possibly multithreaded) transaction executes contiguously, without interleaved operations from outside that transaction. A trace is *externally serializable* if it is equivalent to an externally serial trace.

For the interesting special case when the `main` method of an application is annotated as `deterministic`, external serializability becomes a trivial property (since the execution contains only one transaction), but conflict freedom provides a strong determinism guarantee–that the program will behave the same regardless of how its threads are scheduled.[1] This special case of entirely deterministic applications was also addressed by the Cilk Nondeterminator [7], whereas this paper addresses the problem in a more general setting.

Another interesting case is when a `deterministic` method does not fork additional threads, and so conflict freedom becomes trivial (since the transaction contains only a single thread) and external serializability reduces to the traditional notion of serializability or atomicity. Thus, `deterministic` can be viewed as a generalization of `atomic` that better supports deterministically-parallel computations such as quicksort.

In the more general situation, a program execution may consist of multiple (possibly concurrent) transactions, each of which is internally multithreaded, and the above two correctness properties control thread interference both within and between transactions.

*SingleTrack.* This paper presents a dynamic analysis for verifying conflict freedom and external serializability. To verify conflict freedom, the analysis employs clock vectors [14] as a compact representation of the happens-before relation, and it uses additional mechanisms to track the current transaction for each thread and to distinguish intra-transaction conflicts (which are forbidden) from inter-transaction conflicts (which are allowed). To verify external serializability, the analysis dynamically constructs a *transactional happens-before* graph [10]. This graph encodes which transactions have operations that happen before operations of other transactions, and it contains a cycle if and only if the observed trace violates external serializability.

Figure 2 contains two code fragments that illustrate common patterns for deterministic parallelism found in programs. In the left column, the `main` method starts three concurrent invocations of the `worker` method, where each `worker` invocation repeatedly reads shared data, blocks on a barrier, and then updates disjoint portions of that shared data. The barrier synchronization ensures the absence of conflicts on the reads and writes of the shared data. Although `main` is not `atomic`, our analysis verifies that it is `deterministic`. The right column of Figure 2 shows an idealized implementation of thread pools, in which the assignment of tasks from the work list to worker threads is scheduler-dependent and so non-deterministic. If a program uses a thread pool to execute tasks with `deterministic run` methods, our analysis will still verify that these tasks are deterministic, despite the non-determinism at the application level.

[1] This property assumes that thread scheduling is the only source of non-determinism.

**Figure 2: Common idioms for Deterministic Parallelism**

```
Barrier barrier = new Barrier(3);
int a[] = new int[3];

deterministic void main() {
  fork { worker(0); }
  fork { worker(1); }
  worker(2);
}

void worker(int id) {
  for (int i = 0; i < 10; i++) {
    int tmp = f(a[0],a[1],a[2]);
    barrier.await();
    a[id] = tmp;
    barrier.await();
  }
}
```

(a) Barrier synchronization

```
class ThreadPool {
  BlockingQueue<Runnable> workList
       = new BlockingQueue<Runnable>();

  ThreadPool(int numWorkers) {
    for (int i = 0; i < numWorkers; i++) {
      fork {
        while (true) {
          workList.dequeue().run();
        }
      }
    }
  }

  void execute(Runnable task) {
      workList.enqueue(task);
  }
}
```

(b) Thread pools

We have developed a prototype implementation, called SingleTrack, of this dynamic analysis. Experimental results show that SingleTrack provides a significant improvement over prior atomicity checkers, largely because `deterministic` is a more general non-interference specification than `atomic` and so is applicable to more methods. In effect, this permits us to check more complex code with fewer false alarms than existing tools.

For example, the `sor` benchmark [1] includes six methods that are not atomic because they involve barrier synchronization along the lines shown in Figure 2(a). Atomicity checkers provide no insight regarding thread interference problems in these methods and, in fact, mask a subtle synchronization defect detected by SingleTrack. (The barrier implementation incorrectly relied on writes to a `long` variable being atomic.) After fixing that bug, SingleTrack verified the entire `sor` benchmark as deterministic, whereas Velodrome, a dynamic atomicity checker [10], still reported spurious atomicity violations on the six methods. In addition, SingleTrack verified as deterministic many other problematic non-atomic methods in our benchmarks. Despite its increased generality, SingleTrack's performance is competitive with existing atomicity checkers.

*Contributions.* In summary, this paper:

– identifies a limitation of atomicity for reasoning about the common idiom of deterministic parallelism;
– proposes `deterministic` as a concise specification for this concurrency idiom that combines conflict freedom and external serializability;
– develops a dynamic analysis for verifying this non-interference specification;
– shows that the analysis reports an error whenever the observed trace violates this specification;
– presents an implementation for multithreaded Java programs; and
– validates the effectiveness and performance on a collection of benchmarks.

## 2 Semantics of Multithreaded Programs

To provide a sound basis for our dynamic analysis, we begin by formalizing the semantics of multithreaded programs, as summarized in Figure 3. A program consists of a number of concurrently executing threads that manipulate variables $x \in \mathit{Var}$ and locks $m \in \mathit{Lock}$. Each thread has a thread identifier $t \in \mathit{Tid}$. A program *state* $\Sigma$ maps program variables to values. The state also records the holder (if any) of each lock $m$: if $m$ held by thread $t$, then $\Sigma(m) = t$, and otherwise $\Sigma(m) = \bot$. The state also maps each thread identifier $t$ to a local store $\Sigma(t) = \pi$ for that thread, which contains thread-local data such as the program counter and call stack. The distinguished local stores `NotStarted` and `Finished` indicate threads that have not started running yet and that have finished running, respectively. Execution starts in an initial state $\Sigma_0$, where $\Sigma_0(t) =$ `NotStarted` for all threads $t$ except the initial thread.

*Operations.* Each thread proceeds by performing a sequence of operations on the global store. Thread $t$ can perform all operations $a$ from the following list:

- $\mathit{rd}(t, x, v)$ and $\mathit{wr}(t, x, v)$, which read and write a value $v$ from variable $x$;
- $\mathit{acq}(t, m)$ and $\mathit{rel}(t, m)$, which acquire and release a lock $m$;
- $\mathit{begin}(t)$ and $\mathit{end}(t)$, which demarcate each `deterministic` block;
- $\mathit{fork}(t, u, \pi)$, which forks a new thread $u$ with initial local store $\pi$;
- $\mathit{stop}(t)$, which stops thread $t$; and
- $\mathit{join}(t, u)$, which blocks until thread $t$ terminates via $\mathit{stop}(t)$.

The relation $T(t, \pi, a, \pi')$ holds if the thread $t$ can take a step from a local store $\pi$ to a new local store $\pi'$ by performing the operation $a \in \mathit{Operation}$ on the global store. We assume that $T$ is not defined if either $\pi$ or $\pi'$ is the distinguished local stores `NotStarted` or `Finished`.

The transition relation $\Sigma \rightarrow^a \Sigma'$ performs a single step of execution. It chooses an operation $a$ by thread $t$ that is applicable in the local state $\Sigma(t)$, performs that operation to yield a new local store $\pi'$, and returns a new (appropriately updated) state. An operation $a$ is *enabled* in $\Sigma$ if $\exists \Sigma'$ such that $\Sigma \rightarrow^a \Sigma'$. A state $\Sigma$ is *final* if the local store for every thread in that state is either `NotStarted` or `Finished`. We assume that each operation is deterministic: if $\mathit{tid}(a) = \mathit{tid}(b)$ and $\Sigma \rightarrow^a \Sigma'$ and $\Sigma \rightarrow^b \Sigma''$ then $a = b$ and $\Sigma' = \Sigma''$.

A *trace* $\alpha$ captures an execution of a multithreaded program by listing the sequence of operations performed by the various threads. 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 $\Sigma_0 \rightarrow^{a_1} \Sigma_1 \rightarrow^{a_2} \cdots \rightarrow^{a_n} \Sigma_n$. We assume that each valid trace is cycle free.

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

- **Communication conflict**: they read or write the same variable, and at least one of the accesses is a write.
- **Lock conflict:** they acquire or release the same lock.

**Figure 3: Semantics of Multithreaded Programs**

**Domains:**

$$\begin{array}{rcl} \Sigma \in \mathit{State} & = & (\mathit{Var} \to \mathit{Value}) \\ & & \cup\ (\mathit{Lock} \to \mathit{Tid}_\bot) \\ & & \cup\ (\mathit{Tid} \to \mathit{LocalStore}) \\ & & \\ a \in \mathit{Operation} & ::= & \mathit{rd}(t,x,v) \mid \mathit{wr}(t,x,v) \\ & & \mid \mathit{acq}(t,m) \mid \mathit{rel}(t,m) \\ & & \mid \mathit{begin}(t) \mid \mathit{end}(t) \\ & & \mid \mathit{fork}(t,u,\pi) \mid \mathit{join}(t,u) \mid \mathit{stop}(t) \end{array} \qquad \begin{array}{rl} u,t \in & \mathit{Tid} \\ x \in & \mathit{Var} \\ v \in & \mathit{Value} \\ m \in & \mathit{Lock} \\ \pi \in & \mathit{LocalStore} \end{array}$$

**Transition relation:** $\Sigma \to^a \Sigma'$

[STEP READ]
$$\frac{a = \mathit{rd}(t,x,v) \quad T(t,\Sigma(t),a,\pi') \quad \Sigma(x) = v}{\Sigma \to^a \Sigma[t := \pi']}$$

[STEP WRITE]
$$\frac{a = \mathit{wr}(t,x,v) \quad T(t,\Sigma(t),a,\pi')}{\Sigma \to^a \Sigma[t := \pi', x := v]}$$

[STEP ACQUIRE]
$$\frac{a = \mathit{acq}(t,m) \quad T(t,\Sigma(t),a,\pi') \quad \Sigma(m) = \bot}{\Sigma \to^a \Sigma[t := \pi', m := t]}$$

[STEP RELEASE]
$$\frac{a = \mathit{rel}(t,m) \quad T(t,\Sigma(t),a,\pi') \quad \Sigma(m) = t}{\Sigma \to^a \Sigma[t := \pi', m := \bot]}$$

[STEP BEGIN]
$$\frac{a = \mathit{begin}(t) \qquad T(t,\Sigma(t),a,\pi')}{\Sigma \to^a \Sigma[t := \pi']}$$

[STEP END]
$$\frac{a = \mathit{end}(t) \qquad T(t,\Sigma(t),a,\pi')}{\Sigma \to^a \Sigma[t := \pi']}$$

[STEP FORK]
$$\frac{\begin{array}{cc} a = \mathit{fork}(t,u,\pi'') & T(t,\Sigma(t),a,\pi') \\ \Sigma(u) = \texttt{NotStarted} & \pi' \neq \texttt{NotStarted} \end{array}}{\Sigma \to^a \Sigma[t := \pi', u := \pi'']}$$

[STEP JOIN]
$$\frac{\begin{array}{c} a = \mathit{join}(t,u) \\ T(t,\Sigma(t),a,\pi') \qquad \Sigma(u) = \texttt{Finished} \end{array}}{\Sigma \to^a \Sigma[t := \pi']}$$

[STEP STOP]
$$\frac{a = \mathit{stop}(t) \qquad T(t,\Sigma(t),a,\pi')}{\Sigma \to^a \Sigma[t := \texttt{Finished}]}$$

- **Fork-join conflict**: one operation is $\mathit{fork}(t,u,\pi)$ or $\mathit{join}(t,u)$ and the other operation is by thread $u$.
- **Program order conflict**: they are performed by the same thread.

The *happens-before relation* $<_\alpha$ for a trace $\alpha$ is the smallest transitively-closed relation on operations in $\alpha$ such that if operation $a$ occurs before $b$ in $\alpha$ and $a$ conflicts with $b$, then $a$ *happens-before* $b$.[2]

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.

*Transactions.* A *transaction* in a trace $\alpha$ is the sequence of operations executed by a thread $t$ starting with a $\mathit{begin}(t)$ operation and containing all $t$ operations up to and including a matching $\mathit{end}(t)$ operation. For each operation $\mathit{fork}(t,u,\pi)$ in a transaction, that transaction also includes all operations of the forked thread $u$. Any operation that does not occur within another transaction is considered to execute in its own (unary) transaction. To simplify some aspects of the formal presentation, we assume $\mathit{begin}(t)$ and $\mathit{end}(t)$ operations are appropriately

[2] 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.

matched and are not nested (although our implementation does support nested deterministic specifications). We also assume that all locks acquired within a transaction are released within that transaction.

## 3 Dynamically Verifying Internal Conflict Freedom

We next address how to dynamically verify our notion of conflict freedom, *i.e.*, that each operation in the observed trace does not conflict with any other operation in the same transaction. Thus, for example, a lock acquire should not conflict with any other acquire in the same transaction. Similarly, any read operation in a transaction should not conflict with any write in the same transaction. Note that conflicts between an acquire inside a transaction and an acquire *outside* the transaction are permitted; they may violate external serializability but not conflict freedom.

Our analysis uses clock vectors [14] as a compact representation for the happens-before relation and to identify which operations in a transaction are concurrent. A clock vector $CV : \mathit{Tid} \rightarrow \mathit{Nat}$ maps thread identifiers to clocks. Roughly speaking, if $cv$ is the clock vector for an operation $a$ in a trace, then $cv(t)$ identifies which operations of thread $t$ happen-before that operation $a$ (*i.e.*, those $t$-operations for which $t$'s clock is less than or equal to $cv(t)$).

Clock vectors 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 $\mathit{inc}_t$ increments the $t$-component of a clock vector:

$$\begin{aligned} cv_1 \sqsubseteq cv_2 \quad &\text{iff} \quad \forall t.\ cv_1(t) \leq cv_2(t) \\ cv_1 \sqcup cv_2 \quad &= \quad \lambda t.\ \mathit{max}(cv_1(t), cv_2(t)) \\ c_0 \quad &= \quad \lambda t.\ 0 \\ \mathit{inc}_t(cv) \quad &= \quad \lambda u.\ \texttt{if}\ u = t\ \texttt{then}\ cv(u) + 1\ \texttt{else}\ cv(u) \end{aligned}$$

Our conflict freedom analysis allocates a unique *transaction identifier* $w \in \mathit{Xid}$ for each transaction in the observed trace and records which threads belong to that transaction. The analysis is an online algorithm based on an analysis state $\sigma = (\mathbb{X}, \mathbb{C}, \mathbb{U}, \mathbb{R}, \mathbb{W})$ where:

- $\mathbb{X} : \mathit{Tid} \rightarrow \mathit{Xid}_\perp$ records the current transaction (if any) for each thread;
- $\mathbb{C} : \mathit{Tid} \rightarrow CV$ records the clock vector of the <u>c</u>urrent operation by each thread;
- $\mathbb{U} : \mathit{Lock} \times \mathit{Xid} \rightarrow CV$ records the clock vector of the last <u>u</u>nlock of each lock in each transaction;
- $\mathbb{R} : \mathit{Var} \times \mathit{Xid} \rightarrow CV$ records the join of all clock vectors for all <u>r</u>eads to each variable by each transaction; and
- $\mathbb{W} : \mathit{Var} \times \mathit{Xid} \rightarrow CV$ records the clock vector of the last <u>w</u>rite to each variable in each transaction.

**Figure 4: Dynamically Verifying <u>C</u>onflict <u>F</u>reedom: $\sigma \Rightarrow^a \sigma'$**

[CF BEGIN]

$$\frac{\begin{array}{rcl} \mathbb{X}(t) & = & \bot \\ \mathbb{X}' & = & \mathbb{X}[t := w],\ w \text{ is fresh} \\ \mathbb{C}' & = & \mathbb{C}[t := inc_t(c_0)] \end{array}}{(\mathbb{X}, \mathbb{C}, \mathbb{U}, \mathbb{R}, \mathbb{W}) \Rightarrow^{begin(t)} (\mathbb{X}', \mathbb{C}', \mathbb{U}, \mathbb{R}, \mathbb{W})}$$

[CF END]

$$\frac{\begin{array}{rcl} \mathbb{X}(t) & \neq & \bot \\ \mathbb{X}' & = & \mathbb{X}[t := \bot] \end{array}}{(\mathbb{X}, \mathbb{C}, \mathbb{U}, \mathbb{R}, \mathbb{W}) \Rightarrow^{end(t)} (\mathbb{X}', \mathbb{C}, \mathbb{U}, \mathbb{R}, \mathbb{W})}$$

[CF ACQUIRE]

$$\frac{\begin{array}{rcl} \mathbb{X}(t) & = & w \neq \bot \\ \mathbb{U}(m, w) & \sqsubseteq & \mathbb{C}(t) \end{array}}{(\mathbb{X}, \mathbb{C}, \mathbb{U}, \mathbb{R}, \mathbb{W}) \Rightarrow^{acq(t,m)} (\mathbb{X}, \mathbb{C}, \mathbb{U}, \mathbb{R}, \mathbb{W})}$$

[CF RELEASE]

$$\frac{\begin{array}{rcl} \mathbb{X}(t) & = & w \neq \bot \\ \mathbb{U}' & = & \mathbb{U}[(m, w) := \mathbb{C}(t)] \end{array}}{(\mathbb{X}, \mathbb{C}, \mathbb{U}, \mathbb{R}, \mathbb{W}) \Rightarrow^{rel(t,m)} (\mathbb{X}, \mathbb{C}, \mathbb{U}', \mathbb{R}, \mathbb{W})}$$

[CF READ]

$$\frac{\begin{array}{rcl} \mathbb{X}(t) & = & w \neq \bot \\ \mathbb{W}(x, w) & \sqsubseteq & \mathbb{C}(t) \\ \mathbb{R}' & = & \mathbb{R}[(x, w) := \mathbb{R}(x, w) \sqcup \mathbb{C}(t)] \end{array}}{(\mathbb{X}, \mathbb{C}, \mathbb{U}, \mathbb{R}, \mathbb{W}) \Rightarrow^{rd(t,x,v)} (\mathbb{X}, \mathbb{C}, \mathbb{U}, \mathbb{R}', \mathbb{W})}$$

[CF WRITE]

$$\frac{\begin{array}{rcl} \mathbb{X}(t) & = & w \neq \bot \\ \mathbb{W}(x, w) & \sqsubseteq & \mathbb{C}(t) \qquad \mathbb{R}(x, w) \sqsubseteq \mathbb{C}(t) \\ \mathbb{W}' & = & \mathbb{W}[(x, w) := \mathbb{C}(t)] \end{array}}{(\mathbb{X}, \mathbb{C}, \mathbb{U}, \mathbb{R}, \mathbb{W}) \Rightarrow^{wr(t,x,v)} (\mathbb{X}, \mathbb{C}, \mathbb{U}, \mathbb{R}, \mathbb{W}')}$$

[CF FORK]

$$\frac{\begin{array}{rcl} \mathbb{X}' & = & \mathbb{X}[u := \mathbb{X}(t)] \\ \mathbb{C}' & = & \mathbb{C}[t := inc_t(\mathbb{C}(t)), u := inc_u(\mathbb{C}(t))] \end{array}}{(\mathbb{X}, \mathbb{C}, \mathbb{U}, \mathbb{R}, \mathbb{W}) \Rightarrow^{fork(t,u,\pi)} (\mathbb{X}', \mathbb{C}', \mathbb{U}, \mathbb{R}, \mathbb{W})}$$

[CF JOIN]

$$\frac{\mathbb{C}' = \mathbb{C}[t := \mathbb{C}(t) \sqcup \mathbb{C}(u)]}{(\mathbb{X}, \mathbb{C}, \mathbb{U}, \mathbb{R}, \mathbb{W}) \Rightarrow^{join(t,u)} (\mathbb{X}, \mathbb{C}', \mathbb{U}, \mathbb{R}, \mathbb{W})}$$

[CF STOP]

$$\frac{}{(\mathbb{X}, \mathbb{C}, \mathbb{U}, \mathbb{R}, \mathbb{W}) \Rightarrow^{stop(t)} (\mathbb{X}, \mathbb{C}, \mathbb{U}, \mathbb{R}, \mathbb{W})}$$

[CF OUTSIDE]

$$\frac{\begin{array}{c} \mathbb{X}(t) = \bot \\ a \in \{acq(t,m), rel(t,m), rd(t,x,v), wr(t,x,v)\} \end{array}}{(\mathbb{X}, \mathbb{C}, \mathbb{U}, \mathbb{R}, \mathbb{W}) \Rightarrow^{a} (\mathbb{X}, \mathbb{C}, \mathbb{U}, \mathbb{R}, \mathbb{W})}$$

In the initial analysis state, no thread is in a transaction and all clock vectors are initialized to $c_0$, except each $\mathbb{C}(t)$ starts at $inc_t(c_0)$ to reflect that the first steps by different threads are not ordered.

$$\sigma_0 \quad = \quad (\lambda t.\, \bot, \quad \lambda t.\, inc_t(c_0), \quad \lambda(m, w).\, c_0, \quad \lambda(x, w).\, c_0, \quad \lambda(x, w).\, c_0)$$

The relation $\sigma \Rightarrow^a \sigma'$ is defined in Figure 4. The first rule [CF BEGIN] for *begin*$(t)$ records that thread $t$ is in a fresh transaction, and resets the clock vector for $t$. The complementary rule for *end*$(t)$ records that $t$ is no longer in a transaction. The rule [CF ACQUIRE] checks that each lock acquire happens after the last acquire of that lock in the same transaction. If this check fails, then no rule is applicable and the analysis reports a violation of conflict freedom. Rules [CF READ] and [CF WRITE] check in a similar manner that reads and writes do not conflict with other operations in the same transaction. We update clock vectors for fork and join operations that perform real (non-redundant) synchronization. The rule [CF FORK] for *fork*$(t, u, \pi)$ performs one "clock tick" for threads $t$ and $u$, and [CF JOIN] records that a join operation happens-after the last operation (*i.e.*, the stop operation) of the joined thread. Finally, operations outside a transaction are irrelevant and are ignored via [CF OUTSIDE].

We extend the relation $\sigma \Rightarrow^a \sigma'$ from operations to traces in the expected manner: the relation $\sigma_0 \Rightarrow^\alpha \sigma_n$ holds for a trace $\alpha = a_1.\cdots.a_n$ if there exist intermediate analysis states $\sigma_1, \ldots, \sigma_{n-1}$ such that $\sigma_0 \Rightarrow^{a_1} \sigma_1 \Rightarrow^{a_2} \cdots \Rightarrow^{a_n} \sigma_n$.

*Correctness.* The following lemma summarizes the non-interference guarantee ensured by this analysis. If the entire program trace lies within a single transaction, then conflict freedom guarantees determinism. That is, we can generalize from a single observed trace of the target program to reason about behavior and correctness of all possible traces for that program (assuming of course no sources of non-determinism other than thread scheduling).

**Lemma 1 (Single Transaction Determinism).** *Suppose $\Sigma_0 \to^{\alpha} \Sigma$ where $\Sigma$ is final and $\sigma_0 \Rightarrow^{\alpha} \sigma$ and $\alpha$ contains a single transaction. Then for any other trace $\Sigma_0 \to^{\beta} \Sigma'$ where $\Sigma'$ is final, we have that $\Sigma = \Sigma'$.*

## 4 Dynamically Verifying External Serializability

We next describe our dynamic analysis for the second non-interference property of external serializability. Our analysis allocates a *Node* for each transaction in the observed trace. Then, for each operation in the trace that conflicts with a preceding operation from a different transaction, our analysis adds a directed edge between the nodes for these two transactions. Thus, the analysis computes the *transactional happens-before relation*, where transaction $A$ *happens-before* transaction $B$ in $\alpha$ (written $A \lessdot_{\alpha} B$) if there exists some operations $a$ of $A$ and $b$ of $B$ such that $a <_{\alpha} b$. Then $\alpha$ is serializable if and only if the transactional happens-before order $\lessdot_{\alpha}$ is acyclic. This analysis generalizes the approach used to identify atomicity violations in the VELODROME atomicity checker [10].

Our external-serializability analysis is an online algorithm that maintains an analysis state $\phi = (\mathcal{C}, \mathcal{L}, \mathcal{U}, \mathcal{R}, \mathcal{W}, \mathcal{H})$ where:

- $\mathcal{C} : \mathit{Tid} \to \{\textsc{In}, \textsc{Out}\}$ identifies whether a thread is currently in a transaction;
- $\mathcal{L} : \mathit{Tid} \to \mathit{Node}_{\perp}$ identifies the transaction that executed the last operation (if any) of each thread;
- $\mathcal{U} : \mathit{Lock} \to \mathit{Node}_{\perp}$ identifies the last transaction (if any) to unlock each lock;
- $\mathcal{R} : \mathit{Var} \times \mathit{Tid} \to \mathit{Node}_{\perp}$ identifies the last transaction of each thread to read from each variable;
- $\mathcal{W} : \mathit{Var} \to \mathit{Node}_{\perp}$ identifies the last transaction (if any) to write to each variable; and
- $\mathcal{H} \subseteq \mathit{Node} \times \mathit{Node}$ 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.)

In the initial analysis state $\phi_0$, these components are all empty:

$$\phi_0 \;=\; (\lambda t.\textsc{Out},\;\; \lambda t.\perp,\;\; \lambda m.\perp,\;\; \lambda(x,t).\perp,\;\; \lambda x.\perp,\;\; \emptyset)$$

The relation $\phi \Rrightarrow^{a} \phi'$ shown in Figure 5 updates the analysis state for each operation $a$ of the target program. The first rule [XS BEGIN] for *begin*$(t)$ uses the operation $\mathcal{H} \uplus E$ to extend the happens-before graph with additional edges $E \subseteq \mathit{Node}_{\perp} \times \mathit{Node}_{\perp}$, filtering out self-edges and edges that start or end on $\perp$:

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

**Figure 5: Dynamically Verifying E<u>x</u>ternal-<u>S</u>erializability: $\phi \Rrightarrow^a \phi'$**

In all rules, $\phi = (\mathcal{C}, \mathcal{L}, \mathcal{U}, \mathcal{R}, \mathcal{W}, \mathcal{H})$.

[XS BEGIN]
$$\frac{\begin{array}{rcl} \mathcal{C}(t) &=& \textsc{Out} \\ \mathcal{C}' &=& \mathcal{C}[t := \textsc{In}] \\ \mathcal{L}' &=& \mathcal{L}[t := n],\ n \text{ is fresh} \\ \mathcal{H}' &=& \mathcal{H} \uplus \{\, (\mathcal{L}(t), n)\} \end{array}}{\phi \Rrightarrow^{begin(t)} (\mathcal{C}', \mathcal{L}', \mathcal{U}, \mathcal{R}, \mathcal{W}, \mathcal{H}')}$$

[XS END]
$$\frac{\begin{array}{rcl} \mathcal{C}(t) &=& \textsc{In} \\ \mathcal{C}' &=& \mathcal{C}[t := \textsc{Out}] \end{array}}{\phi \Rrightarrow^{end(t)} (\mathcal{C}', \mathcal{L}, \mathcal{U}, \mathcal{R}, \mathcal{W}, \mathcal{H})}$$

[XS ACQUIRE]
$$\frac{\mathcal{C}(t) = \textsc{In} \qquad \mathcal{H}' = \mathcal{H} \uplus \{(\mathcal{U}(m), \mathcal{L}(t))\}}{\phi \Rrightarrow^{acq(t,m)} (\mathcal{C}, \mathcal{L}, \mathcal{U}, \mathcal{R}, \mathcal{W}, \mathcal{H}')}$$

[XS RELEASE]
$$\frac{\mathcal{C}(t) = \textsc{In} \qquad \mathcal{U}' = \mathcal{U}[m := \mathcal{L}(t)]}{\phi \Rrightarrow^{rel(t,m)} (\mathcal{C}, \mathcal{L}, \mathcal{U}', \mathcal{R}, \mathcal{W}, \mathcal{H})}$$

[XS READ]
$$\frac{\begin{array}{rcl} \mathcal{C}(t) &=& \textsc{In} \\ \mathcal{H}' &=& \mathcal{H} \uplus \{\, (\mathcal{W}(x), \mathcal{L}(t))\} \\ \mathcal{R}' &=& \mathcal{R}[(x,t) := \mathcal{L}(t)] \end{array}}{\phi \Rrightarrow^{rd(t,x,v)} (\mathcal{C}, \mathcal{L}, \mathcal{U}, \mathcal{R}', \mathcal{W}, \mathcal{H}')}$$

[XS WRITE]
$$\frac{\begin{array}{c} \mathcal{C}(t) = \textsc{In} \\ \mathcal{W}' = \mathcal{W}[x := \mathcal{L}(t)] \\ \mathcal{H}' = \mathcal{H} \uplus \{(\mathcal{W}(x), \mathcal{L}(t)),\ (\mathcal{R}(x,u), \mathcal{L}(t)) \mid u \in \mathit{Tid}\} \end{array}}{\phi \Rrightarrow^{wr(t,x,v)} (\mathcal{C}, \mathcal{L}, \mathcal{U}, \mathcal{R}, \mathcal{W}', \mathcal{H}')}$$

[XS FORK IN]
$$\frac{\begin{array}{c} \mathcal{C}(t) = \textsc{In} \\ \mathcal{L}' = \mathcal{L}[u := \mathcal{L}(t)] \qquad \mathcal{C}' = \mathcal{C}[u := \textsc{In}] \end{array}}{\phi \Rrightarrow^{fork(t,u,\pi)} (\mathcal{C}', \mathcal{L}', \mathcal{U}, \mathcal{R}, \mathcal{W}, \mathcal{H})}$$

[XS FORK OUT]
$$\frac{\begin{array}{c} \mathcal{C}(t) = \textsc{Out} \quad n \text{ is fresh} \quad \mathcal{L}' = \mathcal{L}[t := n, u := n] \\ \mathcal{C}' = \mathcal{C}[u := \textsc{Out}] \qquad \mathcal{H}' = \mathcal{H} \uplus \{\, (\mathcal{L}(t), n)\} \end{array}}{\phi \Rrightarrow^{fork(t,u,\pi)} (\mathcal{C}', \mathcal{L}', \mathcal{U}, \mathcal{R}, \mathcal{W}, \mathcal{H}')}$$

[XS JOIN]
$$\frac{\mathcal{C}(t) = \textsc{In} \qquad \mathcal{H}' = \mathcal{H} \uplus \{(\mathcal{L}(u), \mathcal{L}(t))\}}{\phi \Rrightarrow^{join(t,u)} (\mathcal{C}, \mathcal{L}, \mathcal{U}, \mathcal{R}, \mathcal{W}, \mathcal{H}')}$$

[XS STOP]
$$\frac{\mathcal{C}(t) = \textsc{In}}{\phi \Rrightarrow^{stop(t)} (\mathcal{C}, \mathcal{L}, \mathcal{U}, \mathcal{R}, \mathcal{W}, \mathcal{H})}$$

[XS OUTSIDE]
$$\frac{\begin{array}{c} \mathcal{C}(t) = \textsc{Out} \\ a \in \{acq(t,m), rel(t,m), rd(t,x,v), wr(t,x,v), join(t,u), stop(t)\} \\ \phi \Rrightarrow^{begin(t)} \phi_1 \qquad \phi_1 \Rrightarrow^{a} \sigma_2 \qquad \phi_2 \Rrightarrow^{end(t)} \phi' \end{array}}{\phi \Rrightarrow^{a} \phi'}$$

Thus, in [XS BEGIN], 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 [XS ACQUIRE] for $acq(t, m)$ updates the happens-before graph with an edge from the last release $\mathcal{U}(m)$ of that lock. Conversely, [XS RELEASE] for $rel(t, m)$ updates $\mathcal{U}(m)$ with the current transaction.

The rule [XS WRITE] for $wr(t, x, v)$ records that this write happens-after all previous accesses to $x$, and updates $\mathcal{W}(x)$ to denote the current transaction. The rule [XS READ] for $rd(t, x, v)$ records that this read happens-after the last write to $x$, and records that the last read to this variable by this thread is the current transaction. For a fork operation within a transaction, the rule [XS FORK IN] records that the forked thread also executes within that transaction. For forks outside a transaction, [XS FORK OUT] creates a fresh unary transaction $n$ for the fork operation. For other operations outside a transaction, [XS OUTSIDE] enters a new transaction, performs that operation, and then exits that (unary) transaction. We extend the relation $\phi \Rrightarrow^a \phi'$ from operations to traces.

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

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

Our dynamic analysis is sound and in that it identifies exactly those traces that are not externally serializable.

**Lemma 2 (External Serializability).** *Suppose $\Sigma_0 \rightarrow^\alpha \Sigma$ and $\phi_0 \Rrightarrow^\alpha \phi$. Then $\alpha$ is externally serializable if and only if $\phi \notin \mathit{Error}$.*

The preceding lemmas characterize the correctness guarantee provided by each of the conflict-freedom and external-serializability analyses. We now describe how the combination of these two analyses provides a determinism guarantee for programs with multiple transactions (each of which may be internally multithreaded).

The *begin-order* of a serial trace is simply the projection of *begin* operations in that trace, which identifies the order in which the transactions execute while ignoring internal scheduling within each transaction.

$$\mathit{begin\text{-}order}(\alpha) \;=\; \text{projection of } \mathit{begin} \text{ operations in } \alpha\text{, where } \alpha \text{ is serial}$$

We say that two serializable traces $\alpha$ and $\beta$ have the *same commit order* if $\alpha$ and $\beta$ have equivalent serial traces $\alpha'$ and $\beta'$ respectively, such that

$$\mathit{begin\text{-}order}(\alpha') \;=\; \mathit{begin\text{-}order}(\beta')$$

Suppose that $\alpha$ is a program trace that satisfies our analyses. Clearly, a different schedule $\beta$ of the various transactions could change the program's behavior and, for example, cause it to execute code not covered by our analyses. However, if $\beta$ is a serializable trace that has the same commit order as $\alpha$, then $\beta$ is guaranteed to terminate in the same final state as $\alpha$, and thus yield the same observable behavior (where we assume all observations are made by inspecting this final state).

**Theorem 1 (Determinism).** *Suppose $\Sigma_0 \rightarrow^\alpha \Sigma$ and $\sigma_0 \Rrightarrow^\alpha \sigma$ and $\phi_0 \Rrightarrow^\alpha \phi$ where $\Sigma$ is a final state and $\phi \notin \mathit{Error}$. Then any serializable trace that has the same commit order as $\alpha$ will terminate in the same final state.*

## 5 Implementation and Evaluation

We have developed a prototype implementation, called SingleTrack, of our dynamic analysis for deterministic parallelism. The analysis takes as input a Java bytecode program and a specification of which methods should be deterministic. It then monitors program execution and reports a warning whenever a determinism specification is violated. For a conflict freedom error, SingleTrack identifies the two operations within a transaction that conflict. For an external serializability error, SingleTrack identifies the corresponding cycle in the transactional happens-before graph.

SingleTrack is implemented as a component in RoadRunner, a framework we have designed for developing dynamic analyses for multithreaded software. RoadRunner is written entirely in Java and runs on any standard JVM. RoadRunner inserts instrumentation code into the target bytecode program at load time. This code generates a stream of events for lock acquires and releases, field and array accesses, method entries and exits, etc. Back-end tool components, such as SingleTrack, process this event stream as it is generated. Re-entrant lock acquires and releases (which are redundant) are filtered out by RoadRunner to simplify these analyses.

Our SingleTrack implementation extends the analysis described so far in a number of respects, including by supporting additional synchronization primitives such as barriers and semaphores. It also supports nested deterministic blocks. When a determinism error is identified, the tool reports a warning for each deterministic block being violated, and so a single bug may lead to multiple determinism warnings. It also includes a fast happens-before analysis to verify that all array elements and non-`volatile` fields are accessed in a race-free manner. Hence, only synchronization operations and accesses to `volatile` fields must be analyzed for conflict freedom and external serializability, which significantly improves SingleTrack's performance.

We have applied SingleTrack to eight JavaGrande [1] benchmarks (`crypt`, `lufact`, `series`, `sor`, `sparse`, `moldyn`, `montecarlo`, and `raytracer`), `hedc` (a query engine that downloads astronomical data from the web [23]), and four additional programs written by us: `quicksort`, which recursively quicksorts an array, spawning new threads for the recursive calls; `matrixmultiply`, which implements a multithreaded, divide-and-conquer matrix multiplication; `queue-mm`, which uses a thread pool and work queue to perform a number of matrix multiplies simultaneously; and `queue-jg`, which uses a thread pool and work queue to execute the first five JavaGrande benchmarks. All JavaGrande benchmarks were configured to use the small data size and four threads, `hedc` was configured to use four worker threads, and the thread pool programs were configured to use pools with two worker threads.

We performed all experiments on an Apple Mac Pro with dual quad-core 3GHz Pentium Xeon processors and 4GB of memory, using OS X 10.5 and Sun's Java HotSpot Client VM, version 1.5.7. All classes loaded by the benchmark programs were instrumented, except those from the standard Java libraries.

Table 1 presents the size, number of threads, and uninstrumented base running time of each program, as well as the slowdown (as a ratio to the base time) of each program when checked by three dynamic analyses: EmptyTool (which does no work and simply measures the instrumentation overhead), SingleTrack, and the Velodrome atomicity checker [10]. Both SingleTrack and Velodrome used the same fast happens-before race detector mentioned above to avoid the overhead of analyzing race-free data accesses. The average slowdowns for these three tools are 4.3, 10.4, and 10.3, respectively, indicating

| Program | Size (lines) | Num. Threads | Base Time (sec) | Slowdown Empty Tool | Slowdown Single-Track | Slowdown Velo-drome | Velodrome Atomicity Warnings | SingleTrack Deterministic Warnings |
|---|---|---|---|---|---|---|---|---|
| `crypt` | 1,241 | 7 | 0.3 | 3.6 | 18.5 | 18.9 | 4 | 0 |
| `lufact` | 1,627 | 4 | 0.2 | 6.9 | 15.3 | 15.2 | 5 | 0 |
| `series` | 967 | 4 | 2.0 | 1.3 | 1.2 | 1.4 | 4 | 0 |
| `sor` | 876 | 4 | 0.2 | 3.8 | 7.7 | 7.5 | 6 | 6 |
| `sparse` | 868 | 4 | 0.3 | 7.7 | 24.6 | 24.4 | 4 | 0 |
| `moldyn` | 1,402 | 4 | 0.7 | 5.1 | 18.6 | 16.2 | 6 | 0 |
| `montecarlo` | 2,669 | 4 | 1.6 | 2.2 | 6.7 | 6.9 | 5 | 0 |
| `raytracer` | 1,970 | 4 | 0.9 | 13.3 | 19.5 | 19.9 | 5 | 1 |
| `matrixmult` | 301 | 7 | 0.04 | 4.1 | 5.8 | 6.0 | 5 | 0 |
| `quicksort` | 292 | 29 | 0.05 | 4.2 | 5.9 | 5.8 | 5 | 0 |
| `hedc` | 6,400 | 6 | 25.9 | 1.0 | 1.0 | 1.0 | 0 | 0 |
| `queue-jg` | 3,906 | 9 | 4.1 | 2.1 | 9.6 | 10.0 | 28 | 0 |
| `queue-mm` | 449 | 11 | 1.0 | 1.3 | 1.3 | 1.3 | 7 | 0 |

**Table 1: Benchmark Programs.**

that SingleTrack does not introduce much additional overhead over Velodrome, despite checking a more complex non-interference property.

The first ten programs in the table use various fork-join, barrier, and divide-and-conquer idioms, and were designed to be deterministic. For these benchmarks, all methods were specified as `deterministic` for SingleTrack and `atomic` for Velodrome. Experiments using Velodrome produced 49 reports of non-atomic methods. Further inspection revealed that these methods were never intended to be atomic, however, since they involve multithreaded subcomputations. Thus, Velodrome is essentially enforcing the wrong non-interference specification. Consequently, Velodrome provides no useful information about the correctness of these methods. In contrast, SingleTrack eliminates all warnings except those caused by two programming errors: `raytracer` has a known race condition on a checksum field that causes nondeterminism, and `sor` contains a barrier implementation that assumes operations on `long` values are atomic. Fixing these two errors enables SingleTrack to verify that all ten programs are deterministic.

The last three programs submit jobs to a work queue. As illustrated in Figure 2(b), concurrent worker threads introduce non-determinism. Velodrome could verify only that the `hedc` tasks were atomic, but reported atomicity violations for the tasks in `queue-jg` and `queue-mm`. In contrast, SingleTrack successfully verified that the tasks in all three of these benchmarks were deterministic.

To summarize, SingleTrack can verify important non-interference properties for programs that are not supported by current checkers. This greatly reduces the burden on the programmer by eliminating spurious warnings that would otherwise have to be examined manually. In the programs studied, only 10% of the warnings reported by Velodrome reflect real interference errors, whereas all of the SingleTrack warnings reflected real synchronization errors.

## 6 Related Work

Netzer and Miller [16] provide a good overview of various kinds of thread inference errors in multithreaded programs. Much previous work has addressed dynamically detecting race conditions, including via race detectors based on the happens-before relation [4, 20, 5] as well as via extensions of Eraser's lockset algorithm [19], for example, to object-oriented languages [23] and for improved precision or performance [3, 17]. Dynamic race detectors have also been developed for other settings, including for nested fork-join parallelism [15].

A variety of tools have been developed to detect atomicity violations, both statically and dynamically. The Atomizer [8] uses Lipton's theory of reduction [13] to check serializability. Wang and Stoller developed more precise *commit-node* algorithms that address both conflict-atomicity (referred to as atomicity in this paper) and view-atomicity [24].

The Cilk project investigated verifying determinism of entire multithreaded applications, first addressing a more restricted fork-join concurrency structure [7] and later extending that approach to more general locking idioms [2]. While successful for deterministic Cilk applications, this approach does not support applications (like `hedc`, `queue-jg`, and `queue-mm`) that are non-deterministic but contain deterministic subcomputations.

Lightweight transactions (see *e.g.* [21, 11, 12, 22]) offer an interesting alternative to explicit concurrency control, and we believe that a combination or synthesis of these two approaches may yield an attractive programming model. In particular, language runtimes could implement determinism via techniques similar to those used to implement transactions, combined with a deterministic scheduler for threads inside transactions.

Static analyses for verifying atomicity include type systems [9, 18] as well as techniques that look for cycles in the happens-before graph [6]. Compared to dynamic techniques, static systems provide stronger soundness guarantees but typically involve trade-offs between precision and scalability. An interesting topic for future work is the development of static analyses that provide better support for deterministically-parallel software.

## 7 Conclusions

Tools for identifying concurrency errors continue to grow in importance. To be effective, they must be able to verify properties of complex software without burdening the programmer with spurious warning messages. This work attempts to achieve this goal by (1) introducing `deterministic`, a new non-interference specification that generalizes `atomic`, and which provides better support for deterministically-parallel software, and (2) by developing a new sound dynamic analysis to identify `deterministic` specification violations. Experimental results demonstrate the our analysis provides a significant improvement over prior checkers, particularly in terms of its ability to detect bugs and verify non-interference properties for deterministically-parallel software. One avenue for future work is to explore how to best design systems around this property.

This work was supported in part by NSF Grants 0341179, 0341387, 0644130, and 0707885.

## References

1. Java Grande benchmark suite. `http://www.javagrande.org`, 2008.
2. G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark. Detecting data races in Cilk programs that use locks. In *SPAA*, 298–309, 1998.
3. J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridhara. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI*, 258–269, 2002.
4. M. Christiaens and K. D. Bosschere. TRaDe: Data Race Detection for Java. In *International Conference on Computational Science*, 761–770, 2001.
5. T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware Java runtime. In *PLDI*, 245–255, 2007.
6. A. Farzan and P. Madhusudan. Causal atomicity. In *CAV*, 315–328, 2006.
7. M. Feng and C. E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *SPAA*, 1–11, 1997.
8. C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *POPL*, 256–267, 2004.
9. C. Flanagan, S. N. Freund, M. Lifshin, and S. Qadeer. Types for atomicity: Static checking and inference for Java. *TOPLAS*, 30(4):1–53, 2008.
10. C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *PLDI*, 2008.
11. T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA*, 388–402, 2003.
12. T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPOPP*, 48–60, 2005.
13. R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.
14. F. Mattern. Virtual time and global states of distributed systems. In *International Workshop on Parallel and Distributed Algorithms*. 1988.
15. J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Supercomputing*, 24–33, 1991.
16. R. H. B. Netzer and B. P. Miller. What are race conditions? some issues and formalizations. *LOPLAS*, 1:74–88, 1992.
17. E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multihreaded C++ programs. In *PPOPP*, 179–190, 2003.
18. A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller. Automated type-based analysis of data races and atomicity. In *PPOPP*, 83–94, 2005.
19. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *TOCS*, 15(4):391–411, 1997.
20. E. Schonberg. On-the-fly detection of access anomalies. In *PLDI*, 285–297, 1989.
21. N. Shavit and D. Touitou. Software transactional memory. In *PODC*, 204–213, 1995.
22. J. Vitek, S. Jagannathan, A. Welc, and A. L. Hosking. A semantic framework for designer transactions. In *ESOP*, 249–263, 2004.
23. C. von Praun and T. Gross. Object race detection. In *OOPSLA*, 70–82, 2001.
24. L. Wang and S. D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *PPOPP*, 137–146, 2006.