Cooperative Reasoning for Preemptive Execution

Jaeheon Yi Caitlin Sadowski Cormac Flanagan

Computer Science Department
University of California at Santa Cruz
Santa Cruz, CA 95064
{jaeheon,supertri,cormac}@cs.ucsc.edu

Abstract

We propose a *cooperative methodology* for multithreaded software, where threads use traditional synchronization idioms such as locks, but additionally document each point of potential thread interference with a "yield" annotation. Under this methodology, code between two successive yield annotations forms a *serializable transaction* that is amenable to sequential reasoning.

This methodology reduces the burden of reasoning about thread interleavings by indicating only those interference points that matter. We present experimental results showing that very few yield annotations are required, typically one or two per thousand lines of code. We also present dynamic analysis algorithms for detecting *cooperability violations*, where thread interference is not documented by a yield, and for *yield annotation inference* for legacy software.

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

General Terms Languages, Algorithms, Verification

Keywords Atomicity, Yield Annotation, Cooperability, Parallelism, Concurrency

1. Cooperative Concurrency

The widespread adoption of multi-core processors necessitates a software infrastructure that can effectively exploit multiple threads of control. Unfortunately, several decades of experience has demonstrated that developing reliable multithreaded software is problematic at best. We argue the difficulties of multithreaded programming are chiefly due to its *preemptive semantics*: between any two instructions of one thread, interleaved instructions of a second thread could change the state of shared variables, thus influencing the subsequent behavior and correctness of the first thread. We refer to this situation as an *interference point*.

Preemptive semantics fails to identify the interference points that actually impact the behavior of a program: all program points must be considered as potential interference points, until excluded through careful analysis. The pervasive presence of potential interference points invalidates traditional sequential reasoning. To illustrate this difficulty, consider the statement "`x++`". Under sequential semantics, "`x++`" is a simple increment operation, but under preemptive semantics, "`x++`" becomes a *potentially non-atomic read-modify-write*.

To address this problem, we propose a *cooperative methodology* for developing multithreaded software. The central idea behind this methodology is that all thread interference should be explicitly documented with a "yield" annotation. A program is *cooperable* if it satisfies this constraint. Yield annotations are purely for documentation purposes and have no run-time effect. Consequently, this cooperative methodology does not constrain the programmer's ability to use a variety of synchronization idioms (locks, barriers, semaphores, etc) to achieve appropriate parallelism and performance.

The key benefit of explicit yield annotations is that they divide the execution of every thread into *transactions*, each consisting of the sequence of instructions between two successive yield annotations. Since thread interference occurs only at yield annotations, each transaction is serializable and behaves *as if* it is executing serially, without interleaved instructions of other threads. Consequently, the code inside each transaction is amenable to sequential reasoning.

More generally, *sequential reasoning is correct by default*. That is, cooperability guarantees that any section of code unbroken by yield annotations exhibits sequential behavior. In the presence of explicit yield annotations, sequential reasoning naturally generalizes into *cooperative reasoning*, which accounts for explicitly marked thread interference.

Cooperative concurrency decomposes the difficult problem of verifying correctness of a multithreaded program into two simpler subproblems:

1. verifying that the program is cooperable, *i.e.*, that yields document all thread interference;
2. and verifying that the program is correct using cooperative reasoning.

To address subproblem 1, we present COPPER, a dynamic analysis that detects *cooperability violations*: undocumented interference points. COPPER observes the execution trace of the target program, and uses a graph-based algorithm to verify that this observed trace is serializable. In particular, it reports an error if the yield annotations are not sufficient to capture all thread interference.

We present evidence that subproblem 2 is much simpler than verification under traditional preemptive semantics. We propose *interference density* as a metric capturing the difficulty of program reasoning, where the interference density of a program under a particular semantics is the number of interference points per line of code under that semantics. Experimental results show that the inter-

PPoPP'11, February 12–16, 2011, San Antonio, Texas, USA.

	Transaction	Yield
Compile-time analysis	Atomicity	**Cooperability**
Run-time enforcement	Transactional Memory	Automatic Mutual Exclusion

Figure 1. Alternatives to preemptive semantics: one may analyze or enforce either transactions or yields (transactional boundaries).

ference density for the cooperative semantics is at least an order of magnitude lower than for the preemptive or atomic semantics: typically 0.17% of source code lines require yield annotations. Additionally, we present anecdotal evidence that applying cooperability to legacy programs reveals bugs. Across 14 benchmark programs, we found 20 cooperability violations that reflect errors in the code. Finally, a recently completed user study demonstrated that yield annotations are associated with a statistically significant improvement in the ability of programmers to identify synchronization defects [29].

When writing new code, a programmer will co-design the algorithm, its synchronization code, and its yield annotations. For large legacy applications, however, manually writing yield annotations is rather tedious. To address this problem, we present SILVER, a dynamic analysis for inferring yield annotations. If an operation would violate cooperability by causing undocumented thread interference, SILVER automatically inserts an implicit yield right before that operation, thus avoiding the cooperability violation. Due to test coverage issues inherent in any dynamic analysis, SILVER may under-approximate the set of required yields. Nevertheless, we found the inferred yield annotations to be extremely helpful.

Atomicity, Transactional Memory, and AME. Several prior techniques have been proposed for controlling thread interference. Figure 1 summarizes the most closely related proposals, divided along two orthogonal dimensions: whether transactions are analyzed or enforced, and whether one specifies transactions (atomic blocks) or transactional boundaries (yields):

- An atomic block is a code fragment that behaves *as if* it executes serially. While atomicity permits sequential reasoning inside each atomic block, preemptive reasoning is still required outside atomic blocks, and so atomicity requires both kinds of reasoning. In contrast, cooperative reasoning is uniformly applicable to all code under our proposed methodology.
- Transactional memory offers an *enforcement* model of execution for transactions, where the runtime system guarantees that atomic blocks satisfy serializability. However, finding a robust semantics for transactional memory has proven elusive, while performance issues hold back widespread adoption.
- Automatic mutual exclusion (AME) inverts transactional memory by executing all code in a single transaction, unless otherwise specified. We are inspired by AME's feature of safety by default but are focused on analysis not enforcement, since enforcement mechanisms may not be appropriate for legacy programs. Current AME implementations leverage transactional memory implementations, with the problems listed above.

Section 8 contains a more detailed comparison with related work.

Cooperability for Legacy Programs. We believe that our proposed cooperative methodology is well-suited for ongoing maintenance of legacy programs. Understanding legacy programs is often difficult, due to poor documentation and programmer churn. We provide a low-cost migration path for enabling cooperative reasoning in legacy programs: SILVER can provide the initial yield annotations to achieve cooperability, while COPPER can verify cooperability upon subsequent modification.

Note that the inferred yield annotations do not affect the execution semantics of a program. This analysis approach to cooperative semantics distinguishes it from other approaches based on enforcement, such as transactional memory, which explicitly change the execution behavior: transitioning to an enforcement-based approach is potentially much more disruptive, and may unintentionally change the behavior of sensitive legacy applications. Instead, our approach enables cooperative reasoning while retaining the same preemptive execution behavior.

Contributions. The key contributions of this paper are:

- We propose a cooperative methodology for developing multithreaded software.
- We present dynamic analyses for detecting cooperability violations (Section 4) and for inferring yield annotations (Section 5).
- We describe the implementation of these algorithms (Section 6).
- We show that cooperability results in a substantially lower interference density than preemptive semantics or atomicity (Section 7.1).
- We show that across 14 benchmarks, our analyses revealed 20 cooperability violations that are real errors (Section 7.2).

2. Motivating Example

We illustrate the benefits of cooperability using the `Buffer` class defined in Figure 2, which implements a single-element FIFO queue. The class provides blocking `dequeue` and `enqueue` methods, which busy-wait when necessary, along with a `nonBlocking-Dequeue` method, which returns `null` if the queue is empty.

The three columns on the right side of Figure 2 describe where thread interference may occur under the preemptive, atomic, and cooperative semantics, respectively. The preemptive column "P" emphasizes that, in the absence of any analysis, almost every line of code in the `Buffer` class has the potential for concurrent threads to make arbitrary modifications to the program state. Clearly, any attempt to reason about the behavior of this class needs to begin with understanding and limiting where thread interference may occur.

The atomicity column "A" illustrates that the two methods `main` and `nonBlockingDequeue` are not vulnerable to thread inference, since they are atomic. The busy-waiting methods `enqueue` and `dequeue` are not atomic and so are still vulnerable to pervasive thread interference.

We could use syntactic atomic statements to limit where interference may occur in these methods but this approach is unsatisfactory due to the static scope of atomic blocks and their awkward interaction with other control constructs. For example, in the following version of `dequeue`, the atomic blocks clutter the code but are still inadequate; for example, they suggest that interference may occur between the initialization of `elem` and the loop test:

```
Object dequeue() {
  Object elem; atomic { elem = null; }
  while (atomic { elem == null }) {
    atomic { elem = dequeueNonBlocking(); }
  }
  atomic { return elem; }
}
```

Figure 2: Example Program `Buffer`

```
public class Buffer {                                  | P | A | C |
  Object contents = null;                              |   |   |   |
                                                       |   |   |   |
  synchronized Object nonBlockingDequeue() {           |   |   |   |
    Object c = contents;                               | × |   |   |
    contents = null;                                   | × |   |   |
    return c;                                          | × |   |   |
  }                                                    |   |   |   |
                                                       |   |   |   |
  Object dequeue() {                                   |   |   |   |
    Object elem = null;                                | × | × |   |
    while (elem == null) {                             | × | × |   |
      yield;                                           | × | × | × |
      elem = dequeueNonBlocking();                     | × | × |   |
    }                                                  | × | × |   |
    return elem;                                       | × | × |   |
  }                                                    |   |   |   |
                                                       |   |   |   |
  void enqueue(Object o) {                             |   |   |   |
    boolean done = false;                              | × | × |   |
    while (!done) {                                    | × | × |   |
      yield;                                           | × | × | × |
      synchronized (this) {                            | × | × |   |
        if (contents == null) {                        | × | × |   |
          contents = o;                                | × | × |   |
          done = true;                                 | × | × |   |
        }                                              | × | × |   |
      }                                                | × | × |   |
    }                                                  | × | × |   |
  }                                                    |   |   |   |
                                                       |   |   |   |
  public static void main(String[] args) {             |   |   |   |
    final Buffer b = new Buffer();                     | × |   |   |
    fork { b.dequeue(); }                              | × |   |   |
    fork { b.enqueue(1); }                             | × |   |   |
  }                                                    |   |   |   |
}                                                      |   |   |   |
```

Instead of syntactic scoping, one could use `atomic_begin` and `atomic_end` statements to delimit atomic blocks, but this approach is also unsatisfactory. Again taking the `dequeue` example, the first `atomic_begin` and last `atomic_end` statement suggest interference before and after the method call:

```
Object dequeue() {
  atomic_begin;
  Object elem = null;
  while (elem == null) {
    atomic_end;
    atomic_begin;
    elem = dequeueNonBlocking();
  }
  return elem;
  atomic_end;
}
```

If we move these statements to the call sites for dequeue, the nonlocal scoping is awkward. Another problem is that these atomic statements are easy to abuse and misplace; for example, one might put code in between two transactions.

The final cooperative semantics column "C" of Figure 2 shows how yield annotations can precisely and naturally specify thread interference points in both atomic and non-atomic methods. For atomic methods, the absence of yield annotations precludes thread interference. For the two non-atomic methods, interleaved steps of other threads may cause thread interference, but this interference appears *as if* it happens only at the start of each busy-waiting loop, as documented by the yield annotation. Thus, the cooperative semantics needs to consider only one point of thread interference, in contrast to the preemptive or atomic semantics that permit thread interference at all points in these non-atomic methods.

The notion of cooperability also helps detect concurrency errors. For example, consider the following erroneous version of `dequeue`, which busy-waits until the queue is non-empty, and then calls `nonBlockingDequeue`:

```
Object dequeue() {
  while (contents == null) { yield; }
  return nonBlockingDequeue();
}
```

Our analysis would detect that this code is not cooperable, since there is an additional thread interference point at the end of the `while` loop that needs to be explicated via the following additional yield annotation.

```
Object dequeue() {
  while (contents == null) { yield; }
  yield;
  return dequeueNonBlocking();
}
```

This yield annotation now clarifies to the programmer that the buffer state may change between the loop test (`contents == null`) and the call to `dequeueNonBlocking`, and so helps highlight the semantic error in the code, whereby a concurrent thread could remove the buffer contents before `dequeueNonBlocking` is called. In comparison, method-level atomicity would not reveal this error, since the `dequeue` method is intentionally non-atomic.

3. Defining Cooperative Semantics

In order to present our dynamic analyses, we start by formalizing the notion of cooperability in terms of execution traces.

3.1 Core Operations

We are interested in analyzing a running program; an *operation* is the basic entity of analysis during program execution and represents the execution of a single atomic instruction. Our dynamic analyses recognize a concise language of access and synchronization operations: threads t and u may read and write shared variables x, acquire and release mutual exclusion locks m, wait for and notify other threads, and fork a new thread or join (wait) on another thread's completion. These operations represent a core subset of the operations observed during program execution. Volatile variables are simply treated as shared variables in our analyses. Additionally, a yield operation indicates when a yield annotation is seen in the program. These core operations are represented by the grammar:

$$\begin{aligned} op ::= {} & read(t,x,v) \mid write(t,x,v) \\ & \mid acquire(t,m) \mid release(t,m) \\ & \mid prewait(t,m) \mid postwait(t,m) \mid notify(t,m) \\ & \mid fork(t,u) \mid join(t,u) \\ & \mid yield(t) \end{aligned}$$

Some operations are implicitly accompanied by a yield operation. For example, a wait call on lock m releases that lock, waits for another thread to notify m, and then re-acquires m. The execution of a wait call is represented with two operations: $prewait(t,m)$ releases lock m and implicitly may yield to another thread; and $postwait(t,m)$ re-acquires m. Similarly, the *join* operation is preceded by an implicit yield operation, since the joining thread intentionally blocks until the joinee thread completes.

It is straightforward to add atomic block annotations, such as `atomic_begin` and `atomic_end`, to the core language [38]. This extension increases expressivity; *e.g.*, methods may be declared as atomic. However, we omit this extension for clarity of presentation.

3.2 Conflicting Operations

While an operation is the most basic element of analysis, we are more interested in the relationship between operations. Each operation modifies the *program state*, which captures the value of all variables and locks in the running program (including the instruction counter for each thread) at a single point in time.

Two operations that occur in sequence *conflict* if the resulting state may differ when the operations are reordered. There are several kinds of conflicts: writes to a variable conflict with other accesses (reads or writes) to that variable; operations on a lock (acquire, release, prewait, postwait, notify) conflict with each other; forking or joining a thread conflicts with all operations of the target thread; and operations by the same thread conflict with each other. Two operations *commute* if they do not conflict. Either ordering of a commuting pair of operations results in the same state.

A *trace* is a sequence of operations that captures an execution of a running program. We say that two traces are *equivalent* if one can be obtained from the other by applying some number of commutations on adjacent operations in the trace.

The *happens-before* relation [22] orders conflicting operations: we say that a *happens-before* b, or $a < b$, if a and b conflict, and a occurs before b in the trace. The happens-before relation is transitively closed and is a partial order. We use the happens-before relation to characterize how an operation may influence subsequent operations in a trace.

3.3 Preemptive and Cooperative Semantics

A *scheduling semantics* defines the policy for when a context switch may occur in a program. We formalize two scheduling semantics: preemptive and cooperative.

A *preemptive* semantics is one where a context switch may occur after any operation in a trace. Preemptive semantics is the default execution semantics for multithreaded programs. In this semantics, a yield does not influence context switching and thus is a no-op.

Alternatively, in a *cooperative* semantics, context switching may occur either at a yield operation or at thread termination. The yield operations observed for a single thread, in conjunction with the thread's start and end, delimit sequences of operations; each sequence is a *transaction*. A *serial* trace is thus a sequence of transactions. We relate preemptive and serial traces as follows: a trace, perhaps preemptively scheduled, is *serializable* if it is equivalent to a serial trace.

Figure 3 illustrates this notion of a serial trace. In particular, trace A is preemptive and arbitrarily interleaves the operations of the two threads. By repeatedly swapping adjacent commuting operations, however, we can transform trace A into the equivalent trace B. For example, Thread 2's accesses to `done` are local, and thus commute with any operation by Thread 1. The resulting trace B is serial, since it is a sequence of transactions. We then say that the original trace A is serializable, since it is equivalent to the serial trace B.

We say a program is cooperable if it only generates serializable traces. This notion of cooperable programs significantly simplifies reasoning about program correctness. In particular, even though a cooperable program can execute with preemptive semantics on modern multicore hardware, we may use the simpler cooperative semantics to understand the program's behavior. Furthermore, cooperability violations often reveal synchronization errors that result in unintended thread interference (Section 7.2).

Figure 3: Trace of `Buffer.main`

Thread 1	Thread 2
wr elem	
rd elem	
yield	
acq this	
rd contents	
wr c	
wr contents	
rd c	
	wr done
rel this	
	rd done
wr elem	
	yield
rd elem	
	acq this
	rd contents
	rd o
	wr contents
	wr done
	rel this
yield	
acq this	
	rd done
rd contents	
wr c	
wr contents	
rd c	
rel this	
wr elem	
rd elem	
rd elem	
yield	

(A) Observed preemptive trace

Thread 1	Thread 2
wr elem	
rd elem	
yield	
acq this	
rd contents	
wr c	
wr contents	
rd c	
rel this	
wr elem	
rd elem	
yield	
	wr done
	rd done
	yield
	acq this
	rd contents
	rd o
	wr contents
	wr done
	rel this
	rd done
acq this	
rd contents	
wr c	
wr contents	
rd c	
rel this	
wr elem	
rd elem	
rd elem	
yield	

(B) Equivalent serial trace

4. Cooperability Checking

This section presents a dynamic analysis, called COPPER, for detecting cooperability violations by observing a program trace. Conceptually, our analysis maintains a graph structure representing the happens-before relation on transactions, where each node corresponds to some transaction. We abuse terminology slightly to say that transaction e_1 *happens-before* transaction e_2 if e_1 contains an operation that happens-before an operation in e_2. Our analysis checks for a *cycle* between transactions e_1 and e_2, where e_1 happens-before e_2, yet also e_2 happens-before e_1. This type of situation may occur, for example, when operation a in e_1 happens-before an operation b in e_2, and b happens-before an operation c in e_1. Such a cycle implies that the trace is not equivalent to any serial trace, since this cyclic ordering prevents one transaction from being commuted to occur entirely before the other. Our analysis for checking cooperability reports exactly this situation: a *cooperability violation*.

In order to precisely describe and discuss our analysis, we formalize the analysis state as follows. The domain Txn is a set of transactions or nodes in our analysis. Sometimes there does not exist an transaction to refer to, for example, the transaction of the last write for a variable that has not yet been written. We use $\bot$ to denote this situation, and use $Txn_\bot = Txn \cup \{\bot\}$. Our analysis state has 5 components:

- $\mathcal{C}_t$: Txn identifies the current transaction for each thread t. Initially, every thread t starts with a fresh transaction.
- $\mathcal{U}_m$: $Txn_\perp$ identifies the last transaction to release (or unlock) each lock m. Initially, every lock m is not held so $\mathcal{U}_m = \perp$.
- $\mathcal{R}_{x,t}$: $Txn_\perp$ identifies the last transaction of a thread t that reads each variable x. Initially, $\mathcal{R}_{x,t} = \perp$.
- $\mathcal{W}_x$: $Txn_\perp$ identifies the last transaction that wrote to each variable x. Initially, $\mathcal{W}_x = \perp$.
- $\mathcal{H} \subseteq Txn \times Txn$ tracks the transaction-based happens-before relation. For efficiency, transitive edges are not explicitly added to $\mathcal{H}$, and so $\mathcal{H}^*$, the transitive closure of $\mathcal{H}$, represents the happens-before relation on transactions. Initially, we have $\mathcal{H} = \varnothing$.

We present the COPPER algorithm for checking cooperability in Figure 4. COPPER is shown as a function `analyze` that processes each successive operation in a trace, and which has cases for each type of operation. The algorithm is designed to maintain each component in the analysis state for the purpose of accurately tracking the happens-before relation on transactions. In particular, we monitor the $\mathcal{H}$ component for a cycle. Whenever COPPER encounters an operation that conflicts with a previous operation, it tries to add the necessary happens-before edge between transactions to $\mathcal{H}$. Since the happens-before relation is lifted from operations to transactions and the algorithm checks for cycles in this relation, we must take care not to accidentally add self-edges. We use the binary $\uplus$ operator to add a set of edges $E \subseteq Txn_\perp \times Txn_\perp$ to the happens-before graph $\mathcal{H}$ in a safe manner:

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

Three types of operations start a new transaction: *yield*, *prewait*, and *join*. The `analyze` case for *yield* creates a fresh node n, adds the appropriate intrathread edge to $\mathcal{H}$, and updates $\mathcal{C}_t$ with the fresh node. The `analyze` cases for *prewait* and *join* contain a call to `analyze(`$yield(t)$`)` that reflect their implicit yield annotations.

Three types of operations may introduce a cycle in the happens-before graph: *read*, *write*, and *acquire*. These operations all can cause thread interference by racing to access a shared variable or acquire a lock. Figure 5 shows the function `addInEdges`(t, N), which adds an edge from each node $n \in N$ to C_t, and also checks for and handles cooperability violations. When adding an edge that would create a cycle, `addInEdges` detects this situation and does not add that edge, in order to preserve the acyclicity of $\mathcal{H}$; instead, it reports a cooperability violation.

The `analyze` case for $read(t, x, v)$ adds a happens-before edge from $\mathcal{W}_x$, the last transaction to write to x, to the current transaction, via a call to `addInEdges`. Next, $\mathcal{R}_{x,t}$ is updated with the current transaction $\mathcal{C}_t$, reflecting the last read operation of x by t.

Similarly, the `analyze` case for $write(t, x, v)$ adds happens-before edges from $\mathcal{W}_x$ and $\mathcal{R}_{x,t'}$ to $\mathcal{C}_t$, because the write operation conflicts with the prior write and also all prior reads. Next, $\mathcal{W}_x$ is updated with the current transaction $\mathcal{C}_t$, reflecting the last write operation of x.

The `analyze` case for $acquire(t, m)$ processes the happens-before edge between the last release of m and the current acquire operation by t. The corresponding case for $release(t, m)$ simply stores the last transaction that released m.

As discussed in Section 3.1, a *prewait* operation can be viewed as a *release* followed by a *yield*, and a *postwait* operation re-acquires the released lock. The `analyze` case for *notify* is a no-op, since *notify* does not release the lock.

The `analyze` case for $fork(t, u)$ adds a happens-before edge from the forking thread t to the first transaction of the forked thread u, which is always fresh. Similarly, the case for $join(t, u)$ adds a happens-before edge from the last transaction of the joinee thread u to the current transaction of t, the joining thread. However, since we model *join* using an implicit yield, the case includes a call to `analyze(`$yield(t)$`)`.

Figure 4: COPPER Cooperability Checking Algorithm

```
analyze yield(t):
  fresh n
  H ← H ⊎ {(C_t, n)}
  C_t ← n

analyze read(t, x, v):
  addInEdges t {W_x}
  R_{x,t} ← C_t

analyze write(t, x, v):
  addInEdges t {W_x}
  addInEdges t {R_{x,t'} | t' ∈ Tid}
  W_x ← C_t

analyze acquire(t, m):
  addInEdges t {U_m}

analyze release(t, m):
  U_m ← C_t

analyze prewait(t, m):
  analyze release(t, m)
  analyze yield(t)

analyze postwait(t, m):
  analyze acquire(t, m)

analyze notify(t, m):
  no-op

analyze fork(t, u):
  H ← H ⊎ {(C_t, C_u)}

analyze join(t, u):
  analyze yield(t)
  H ← H ⊎ {(C_u, C_t)}
```

Figure 5: `addInEdges` for Detecting Cooperability Violations

```
addInEdges t N =
  if cycle in H ⊎ {(n, C_t) | n ∈ N}
    report cooperability violation
  else
    H ← H ⊎ {(n, C_t) | n ∈ N}
  endif
```

5. Yield Inference

A cooperability checker is useful for verifying that an execution trace is serializable and that thread interference is documented via yield annotations. For legacy programs, however, such yield annotations may not exist, and manually annotating such programs imposes a burden. SILVER, our yield inference algorithm, alleviates this annotation burden.

The SILVER algorithm mostly behaves identically to the COPPER checking algorithm, until a cooperability violation appears. We

Figure 6: `addInEdges` **for Inferring Yield Annotations**

```
addInEdges t N =
  if cycle in H ⊎ {(n, C_t) | n ∈ N} then
    insert a yield just before the current operation
    analyze yield(t)
  endif
  H ← H ⊎ {(n, C_t) | n ∈ N}
```

have modularized these two algorithms so that their difference is isolated to the function `addInEdges`. Figure 6 presents the SILVER version. When the analysis of a particular operation would cause a cooperability violation and associated cycle in the happens-before graph, this function automatically inserts a yield operation right before the current operation *op*. This new yield operation means that *op* now executes in a fresh transaction, and precludes the potential for incoming edges (from nodes in N to that fresh transaction) from forming a cycle.

It is trivial to make any program cooperable: simply add enough yields to the program. Too many yields, however, are counterproductive and noisy. SILVER counteracts noise by minimizing the number of yield annotations inferred, adding one only when necessary. Although we have not yet formalized this notion of minimality, our experiments demonstrate that SILVER infers a small number of yield annotations.

6. Implementation

We implemented the COPPER cooperability checker and SILVER yield inference algorithm using the ROADRUNNER framework [14] for dynamic analysis of multithreaded Java programs. ROADRUNNER dynamically instruments the target bytecode of a program during load time. The instrumentation code creates a stream of events for field and array accesses, synchronization operations, thread fork/join, etc, and the COPPER and SILVER tools operate on this event stream as it is created.

The implementation closely follows the analysis. The happens-before relation on transactions $\mathcal{H}$ is represented by a list of ancestor transactions: if transaction e_1 happens-before transaction e_2, then e_2's ancestor set will contain e_1. The ancestor sets are transitively closed, making cycle detection an $O(1)$ operation.

Scaling the implementation to handle realistic benchmark programs is a key challenge, especially for memory usage. One problem is object churn, where short-lived transactions are created and then immediately thrown away. Object churn puts pressure on the garbage collector, and may adversely impact performance. Instead, SILVER and COPPER statically allocate a set of transaction objects. Every transaction starts `free`, is `inUse` when referenced in the analysis, and becomes `free` when no longer referenced. Reference counting of in-edges determines when an `inUse` transaction becomes `free`: since a completed transaction with no in-edges will never incur additional in-edges in the future, such transactions are ineligible to form a cycle and may be marked as `free`.

Scalability could also suffer if too many transactions are `inUse` simultaneously, saturating the statically allocated set of transactions. The use of weak references solves this issue in practice by allowing significantly more transactions to be considered `free` earlier in the trace. In particular, a transaction n referred to by $\mathcal{W}_x$, $\mathcal{R}_{x,t}$ or $\mathcal{U}_m$ can be involved in a cycle only if some transaction in $\mathcal{C}_t$ or $\mathcal{H}$ refers to n. Our implementation requires reference counting only for $\mathcal{C}_t$ and $\mathcal{H}$, and otherwise uses weak references for $\mathcal{W}_x$, $\mathcal{R}_{x,t}$ and $\mathcal{U}_m$.

7. Evaluation

This section compares the number of interference points for cooperability versus preemptive semantics and atomicity, and evaluates the effectiveness of cooperability at finding synchronization defects. We also briefly discuss the performance of our analysis tools.

We used a collection of multithreaded Java benchmarks ranging in size from 1,000 to 50,000 lines of code. These benchmarks include: `colt`, a library for high performance computing [36]; `hedc`, a warehouse webcrawler for astrophysics data [36]; `raja`, a raytracer program [17]; `elevator`, a real-time discrete event simulator [36]; `mtrt`, a raytracer program from the SPEC JVM98 benchmark suite [32]; and several benchmarks (`crypt`, `lufact`, `moldyn`, `montecarlo`, `raytracer`, `series`, `sor`, `sparse`) from the Java Grande set [20] (configured with four threads and the base data set). Included is a large reactive benchmark: `jigsaw`, W3C's web server [33], coupled with a stress test harness.

Excluding the Java standard libraries, all classes loaded by benchmark programs were instrumented. In all experiments, we used a fine granularity for array accesses, with a distinct shadow object for each element in an array. We ran these experiments on a machine with 12 GB memory and eight cores clocked at 2.8 GHz, running Mac OS X 10.6.1 with Java HotSpot Server VM 1.6.0.

We report the *loaded lines of code* (LLOC) for each benchmark in Column 3 of Figure 7, a line count of Java source files that have actually been loaded by the JVM. The loaded lines of code provide a more accurate impression of a program's size as compared to raw line count, which may include vast amounts of dead code.

7.1 Interference Density

Cooperability greatly reduces the number of interference points in a program, compared to atomicity or preemptive semantics. To substantiate this claim, we collected data about our benchmark set to compare the number of interference points for three distinct semantics:

- In preemptive semantics, every access to a shared variable and every lock acquire is a potential point of interference. We count the number of syntactic program locations corresponding to access and acquire operations seen in the trace. We note that this count already excludes many operations that do not cause interference, such as operations on local variables, lock releases, method calls, etc.
- In *atomic semantics*, a context switch may occur only outside an atomic block; thus, in this semantics, we count thread interference points that occur outside atomic methods. The knowledge of which methods are non-atomic is obtained *a priori* by running an atomicity checker [16]. In addition, when in a non-atomic method, we also count every call to an atomic method, since thread interference may occur just before such a call.
- In cooperative semantics, we report the number of yield annotations inferred by SILVER on unannotated code.

To collect this data, we ran SILVER on unannotated code to obtain a complete set of yield annotations sufficient to guarantee serializability for the observed trace. Because SILVER is a dynamic analysis, it may infer only a subset of the necessary yield annotations for the program. In practice, we find that the inferred yield annotations are a fairly precise underapproximation: COPPER typically verifies subsequent runs over this set of inferred yields. Also, a quick experiment shows that the number of additional yield annotations inferred in later executions diminishes significantly (Figure 8). Note that due to scheduling nondeterminism, COPPER may observe a sufficiently different trace than SILVER, and thus may occasionally report a new cooperability violation.

Program	Thread Count	Size (LLOC)	*Preemptive*		*Atomic*		*Cooperative*		Intended Yields	Errors
			Points	Density	Points	Density	Points	Density		
sparse	4	712	196	27.53%	49	6.88%	1	0.14%	1	0
sor	4	721	134	18.59%	49	6.80%	4	0.55%	3	1
series	4	811	90	11.10%	31	3.82%	1	0.12%	1	0
crypt	7	1083	252	23.27%	55	5.08%	2	0.18%	2	0
moldyn	4	1299	737	56.74%	64	4.93%	4	0.31%	4	0
elevator	5	1447	247	17.07%	54	3.73%	3	0.21%	1	2
lufact	4	1472	242	16.44%	57	3.87%	4	0.27%	4	0
raytracer	4	1862	355	19.07%	65	3.49%	4	0.21%	3	1
montecarlo	4	3557	377	10.60%	41	1.15%	2	0.06%	1	1
hedc	6	6409	305	4.76%	76	1.19%	4	0.06%	3	4
mtrt	5	6460	695	10.76%	25	0.39%	2	0.03%	1	1
raja	2	6863	396	5.77%	45	0.66%	1	0.01%	1	0
colt	11	25644	601	2.34%	113	0.44%	16	0.06%	16	0
jigsaw	77	48674	3415	7.02%	550	1.13%	52	0.11%	34	10
Total/Averages		107014	8042	16.50%	1274	3.11%	100	0.17%		

Figure 7. Interference Points and Interference Density Under Preemptive, Atomic, and Cooperative Semantics

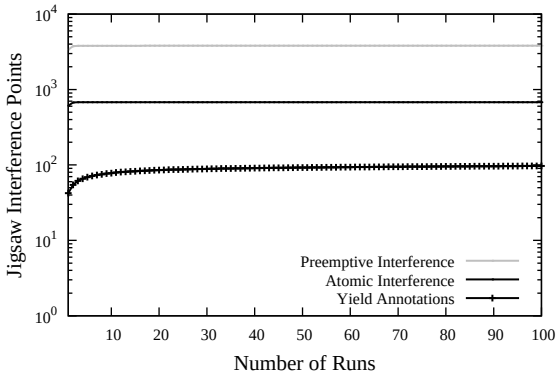

Figure 8. The yield annotations inferred by SILVER for `jigsaw` are stable over multiple runs.

In order to quantify the relative frequency of interference points, we use *interference density*, the number of interference points per loaded line of code. Interference density provides a way to compare the amount of interference across different benchmarks.

Our results are listed in Figure 7, which compares the number of interference points and interference density across our benchmark set according to the three semantics. For preemptive semantics, the interference density ranges from at least 2% to over 50%, and gives a measure of the high cost of reasoning about programs in a preemptive manner. For atomic semantics, the interference density drops by roughly an order of magnitude, but can still reach almost 7% of loaded lines of code, implying that 1 out of every 14 lines must be scrutinized to understand the effects of thread interference, even with method-level atomicity annotations.

For cooperative semantics, the interference density drops by a further order of magnitude as compared with atomic semantics. Indeed, cooperative interference density never rises above 0.6% for our benchmark set, implying that fewer than 6 lines per *thousand* impact the meaning and understandability of a program with respect to nondeterministic scheduling. These results show that cooperability significantly reduces the number of thread interference points to consider in multithreaded programs, as compared to prior techniques. Put differently, yield annotations are relatively rare, and thus do not degenerate into noise.

7.2 Defect Detection

Legacy programs can be difficult to understand, and are often sensitive to change. A set of *intended* yield annotations for such a program assists in understanding thread interference without perturbing preemptive executions. Furthermore, given a legacy program with intended yield annotations, cooperability violations correspond to likely bugs.

Our analysis approach to cooperative semantics can be contrasted with other approaches based on enforcement, such as automatic mutual exclusion, which explicitly change the execution semantics. Transitioning to an enforcement-based approach is potentially disruptive, and may unintentionally change the execution behavior for legacy applications.

To evaluate the effectiveness of cooperability at finding and understanding bugs, we first ran SILVER on our benchmark set and by manual inspection distilled a set of intended yields from the inferred yield annotations, as shown in Column 10 of Figure 7. These intended yields can differ from the inferred ones, due to the choice of yield annotation placement. We then ran COPPER on the resulting yield-annotated program and inspected the results; Column 11 summarizes the number of code bugs that we detected, a number of which reflect known race conditions and destructive atomicity violations.

sparse, series, crypt, raja. The intended thread interference in these programs are all due to the implicit yielding before a join.

sor. For the `sor` benchmark, SILVER reports four yield annotations. One is caused by a join's implicit yield. The other three yield annotations are inferred near code intended to act as a barrier. Of these three, two are intended, while the remaining yield annotation actually reflects an error in the barrier implementation. The barrier in `sor` fails to synchronize accesses to individual non-volatile array elements, and introduces nondeterminism into `sor`.

moldyn, lufact. All of the inferred yield annotations for `moldyn` and `lufact` are situated in barrier code. After manually adding yield annotations before and after barrier operations, COPPER verified the serializability of the observed traces.

elevator. SILVER inferred several yield annotations that implicate two methods as non-atomic (`claimUp` and `claimDown`), both of which have known destructive atomicity violations.

Program	Base Time (ms)	Slowdown EMPTY	Slowdown VELODROME	Slowdown COPPER	Base Memory (MB)	Memory Overhead EMPTY	Memory Overhead VELODROME	Memory Overhead COPPER
colt	16108	0.8	0.8	0.8	34	2.1	3.1	3.9
crypt	321	6.9	20.0	20.8	34	3.7	12.6	13.3
lufact	358	4.5	7.6	7.9	24	1.9	3.1	4.0
moldyn	806	8.8	12.2	12.9	23	2.0	2.8	3.8
montecarlo	1220	5.0	10.6	11.2	116	5.2	8.4	8.0
mtrt	498	7.6	9.4	9.7	48	2.4	4.1	4.5
raja	420	6.0	7.6	7.4	34	1.7	2.2	2.8
raytracer	770	5.4	23.9	16.5	32	1.4	1.9	2.6
series	1111	1.7	2.1	2.1	22	1.9	2.6	3.6
sor	465	3.9	11.3	21.7	35	1.7	3.1	3.7
sparse	457	6.1	16.3	16.9	36	1.8	3.6	4.3
Average		5.2	11.1	11.6		2.3	4.3	5.0

Figure 9. Time and Space Overheads for EMPTY, VELODROME, and COPPER.

raytracer. This program needs four yield annotations for ensuring cooperability: two are in barrier code, one is an implicit yield associated with a join, and one corresponds to a known destructive race on `JGFRayTracerBench.checksum1`.

montecarlo. SILVER inferred two yield annotations, corresponding to a join and a racy variable access.

hedc. The `hedc` benchmark has four racy variables, some of which cause erroneous behavior. After adding a set of intended yields to the program, COPPER issued thread interference reports corresponding to the four racy variables.

mtrt. SILVER reported the implicit yield on a join and inferred a yield annotation for a known racy variable access.

colt. We believe that all the inferred yield annotations in the `colt` benchmark are intentional.

jigsaw. For the `jigsaw` yield annotations, five are caused by waits, and 29 are interference points induced by acquire operations. The remaining 18 inferred yield annotations are caused by 8 racy variables and 2 destructive atomicity violations. For example, thread interference just before a call to `updateStatistics()` in `org.w3c.jigsaw.http.httpd` could result in slightly outdated statistics information. Thread interference between checking whether `log` is null and accessing `log` in `org.w3c.jigsaw.http.CommonLogger` could cause a null pointer exception, if the `shutdown` method runs concurrently.

While finding bugs in code is clearly important, relating how such bugs interact with the rest of the program is just as important for understanding and fixing them. A maintenance strategy based on cooperability improves over existing analyses by providing a richer context for understanding bugs. COPPER's reports cohesively connect race conditions and atomicity violations: thread interference is often caused by a race on a variable or lock, while the call stack at the interference point informs us which methods are non-atomic due to that race.

As an additional data point regarding the benefits of cooperability, we recently conducted a user evaluation on the effectiveness of cooperability for highlighting concurrency bugs [29]. We found that for 2 out of 3 sample programs, yield annotations were associated with a statistically significant improvement in bugs found, compared to un-annotated programs. This study demonstrates the potential of cooperability to significantly improve understanding of thread interference.

7.3 Performance

Figure 9 reports on the performance of our analyses. We compare with the VELODROME atomicity checker [16]. VELODROME is a good choice for comparison, since COPPER and VELODROME check related properties, and both use similar cycle-based techniques. For each of the compute-bound benchmarks, we report on the running time and memory usage of that benchmark (without instrumentation), and on the slowdown and memory overhead incurred by ROADRUNNER when running with the EMPTY tool (which just measures the instrumentation overhead but performs no dynamic analysis); with VELODROME; and with COPPER. We used an implementation of FASTTRACK [13], a modern precise race detector, to filter out non-racy accesses for COPPER and VELODROME; these accesses form the majority of operations in a trace.

The results show that COPPER has an acceptable slowdown and memory overhead for this type of dynamic analysis, and exhibits a similar performance profile to VELODROME. The average slowdown for COPPER was 11.6x, as compared with a slowdown for VELODROME of 11.1x. Similarly, the average memory overhead for COPPER was 5.0x, as compared with a memory overhead for VELODROME of 4.3x. The SILVER inference algorithm exhibits performance similar to the COPPER checking algorithm, which is unsurprising since the two algorithms are closely related.

8. Related Work

Cooperability The notion of cooperability was partly inspired by recent work on automatic mutual exclusion, which proposes ensuring mutual exclusion by default [18, 1]. A major difference is in the meaning of a yield annotation: automatic mutual exclusion *enforces* a yield at runtime, while COPPER *checks* that such a yield annotation guarantees an equivalent serial trace. In prior work, we presented a type and effect system for cooperability [38], which is complementary to the dynamic analysis approach explored in this paper. Kulkarni *et al.* explore *task types*, a data-centric method for obtaining *pervasive atomicity* [21], a notion closely related to cooperability. In their system, threading and data sharing are made explicit via task types, and a combination of type checking and runtime monitoring guarantee correct sharing between threads.

Cooperative semantics have been explored in various settings; for example in high scalability thread packages [35] and as alternative models for structuring concurrency [3, 4, 9]. TIC, a cooperative extension of transactional memory [31], uses a `wait` construct to suspend a transaction and allow conditional interference. Jalbert and Sen focus on simplifying buggy traces by minimizing context switches, which can be understood as providing an equivalent (buggy) trace under a cooperative scheduler [19]. Our work on cooperability is complementary to this work: serializable traces are already partially simplified since context switches need only be considered at yield annotations.

Race Freedom A data race is a well-known situation where two threads simultaneously access a shared variable without synchronization, and at least one thread is writing to that variable. Races often reflect incorrect synchronization, and the absence of races guarantees that a program's behavior can be understood *as if* it is executing on a sequentially-consistent memory model [2].

A large amount of literature has been devoted to finding and fixing data races in an efficient manner: see, for example [13, 27, 8]. However, race-free programs may still exhibit unintended thread interference, because race freedom is a low-level property dealing with memory accesses. That is, higher-level semantic notions of thread non-interference are not addressed by race freedom.

Recent work by Matsakis and Gross address the notion of time in a type system by making execution *intervals* and the happens-before relation between intervals [24] explicit. This approach increases the annotation burden in favor of increased precision and modularity in statically analyzing for data races.

Atomicity A variety of tools have been developed to detect atomicity violations, both statically and dynamically. Static analyses for verifying atomicity include type systems [15, 30] and techniques that look for cycles in the happens-before graph [11]. Compared to dynamic techniques, static systems provide stronger soundness guarantees but typically involve trade-offs between precision and scalability.

Dynamic techniques analyze a specific executed trace at runtime. Artho *et al.* [5] develop a dynamic analysis tool to identify one class of "higher-level races" using the notion of view consistency. Wang and Stoller describe several dynamic analysis algorithms [37]. Sen and Park [26] develop a tool called AtomFuzzer which attempts to schedule an atomicity violation into exhibiting a real error. The closest atomicity analysis is VELODROME [16], which uses a cycle-based algorithm with transactions as nodes. Farzan and Madhusudan [12] provide space complexity bounds for a similar analysis.

Software transactional memory [23] is a concurrency programming model proposed as an alternative to lock-based concurrency, where the runtime system ensures the atomic execution of transactions. Some recent work has been devoted to addressing the semantic difficulties of software transactional memory [31] and making it compatible with lock-based programming [34]. The relationship between atomicity and transactional memory is analogous to that between cooperability and automatic mutual exclusion: software transactional memory enforces the atomic execution of transactions, while atomicity guarantees that atomic blocks always execute as if in such a transaction.

Deterministic Parallelism As other researchers have begun to realize limits of atomicity, there has been renewed focus on checking for deterministic parallelism. SingleTrack [28] goes beyond the notion of atomicity to check serializability of multithreaded transactions which are also verified to be free of internal conflicts (thus guaranteeing determinism at the transaction level). Deterministic Parallel Java [7] introduces a type system to guarantee determinism by default. Burnim et al. present an assertion-based mechanism for checking determinism where bridge assertions can be used to relate different executions of the same program [10]. Enforcement schemes for determinism also exist [25, 6].

9. Future Work

One interesting topic for future work is to explore more expressive yield annotations. For example, a join could yield only to the joinee thread. This way, the use of fork/join parallelism for deterministic computations would not involve the creation of yield annotations. Alternatively, our analysis could be extended to support annotations of data or methods as yielding.

A semantics with both yield and atomic annotations would allow atomic methods to be specified at the interface level and yield annotations to pinpoint why code is not atomic. We feel that this semantics would combine the strengths of each type of annotation.

When inferring yield annotations, there are multiple choices of where to insert a yield annotation, and another area of future work is to insert yield annotations higher in the call stack, for example, at a call to a synchronized method. We are still investigating which annotation placement is the most useful to infer.

Inspired by the results of our prior user study, we would like to investigate some of these upcoming design decisions through more user studies. Also, we are exploring the development of program logics for cooperability, perhaps extending Hoare logic, to help clarify the benefits of cooperative reasoning.

References

[1] M. Abadi and G. Plotkin. A model of cooperative threads. In *Symposium on Principles of Programming Languages (POPL)*, 2009.

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

[3] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *Annual Technical Conference*. USENIX Association, 2002.

[4] R. M. Amadio and S. D. Zilio. Resource control for synchronous cooperative threads. In *International Conference on Concurrency Theory (CONCUR)*, 2004.

[5] C. Artho, K. Havelund, and A. Biere. High-level data races. In *International Conference on Verification and Validation of Enterprise Information Systems (ICEIS)*, 2003.

[6] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe multithreaded programming for C/C++. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2009.

[7] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for Deterministic Parallel Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2009.

[8] M. D. Bond, K. E. Coons, and K. S. McKinley. PACER: Proportional detection of data races. In *Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[9] G. Boudol. Fair cooperative multithreading. In *International Conference on Concurrency Theory (CONCUR)*, 2007.

[10] J. Burnim and K. Sen. Asserting and checking determinism for multithreaded programs. In *International Symposium on Foundations of Software Engineering (FSE)*, 2009.

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

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

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

[14] C. Flanagan and S. N. Freund. The RoadRunner dynamic analysis framework for concurrent programs. In *Program Analysis for Software Tools and Engineering (PASTE)*, 2010.

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

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

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

[18] M. Isard and A. Birrell. Automatic mutual exclusion. In *Workshop on Hot Topics in Operating Systems (HOTOS)*, 2007.

[19] N. Jalbert and K. Sen. A trace simplification technique for effective debugging of concurrent programs. In *International Symposium on Foundations of Software Engineering (FSE)*, 2010.

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

[21] A. Kulkarni, Y. D. Liu, and S. F. Smith. Task types for pervasive atomicity. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2010.

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

[23] J. R. Larus and R. Rajwar. *Transactional Memory*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2006.

[24] N. D. Matsakis and T. R. Gross. A time-aware type system for data-rrace protection and guaranteed initialization. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2010.

[25] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

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

[27] P. Ratanaworabhan, M. Burtscher, D. Kirovski, B. Zorn, R. Nagpal, and K. Pattabiraman. Detecting and tolerating asymmetric races. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2009.

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

[29] C. Sadowski and J. Yi. Applying usability studies to correctness conditions: A case study of cooperability. In *Onward! Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, 2010.

[30] 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 (PPoPP)*, 2005.

[31] Y. Smaragdakis, A. Kay, R. Behrends, and M. Young. Transactions with isolation and cooperation. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007.

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

[33] The World Wide Web Consortium. Jigsaw Web Server. Available from `http://www.w3.org/Jigsaw/`, 2009.

[34] T. Usui, Y. Smaragdakis, R. Behrends, and J. Evans. Adaptive locks: Combining transactions and locks for efficient concurrency. In *Parallel Architectures and Compilation Techniques (PACT)*, 2009.

[35] R. Von Behren, J. Condit, F. Zhou, G. Necula, and E. Brewer. Capriccio: scalable threads for internet services. *ACM SIGOPS Operating Systems Review*, 37(5):281, 2003.

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

[37] 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 (PPoPP)*, 2006.

[38] J. Yi and C. Flanagan. Effects for cooperable and serializable threads. In *Workshop on Types in Language Design and Implementation (TLDI)*, 2010.