Lock Removal for Concurrent Trace Programs [⋆]

Vineet Kahlon[1] and Chao Wang[2]

[1] NEC Laboratories America, Princeton, NJ 08540
[2] Department of ECE, Virginia Tech, Blacksburg, VA 24061

Abstract. We propose a trace-based concurrent program analysis to *soundly* remove redundant synchronizations such as locks while preserving the behaviors of the concurrent computation. Our new method is computationally efficient in that it involves only *thread-local* computation and therefore avoids interleaving explosion, which is known as the main hurdle for scalable concurrency analysis. Our method builds on the partial-order theory and a unified analysis framework; therefore, it is more generally applicable than existing methods based on simple syntactic rules and *ad hoc* heuristics. We have implemented and evaluated the proposed method in the context of runtime verification of multithreaded Java and C programs. Our experimental results show that lock removal can significantly speed up symbolic predictive analysis for detecting concurrency bugs. Besides runtime verification, our new method will also be useful in applications such as debugging, performance optimization, program understanding, and maintenance.

1 Introduction

Concurrent programs are notoriously difficult to analyze due to their behavioral complexity resulting from the often extremely large number of thread interleavings. This renders comprehending all the possible ways in which threads interact a difficult problem. As a result, programmers often take a defensive stance and label large sections of code as critical sections. This may result in the addition of redundant locks, both degrading performance and making program modeling, analysis, and understanding difficult. The situation is particularly severe in trace-based concurrent program analysis. When focusing on a concrete execution trace rather than the entire program, we often find significantly more redundant locks, i.e. locks that are not completely redundant in the whole program may become redundant when the analysis is restricted to a trace.

Although there exist some methods for identifying redundant synchronizations in Java and C programs [3, 4, 6, 22, 1, 30], e.g. as part of the compiler's performance optimization, they are all based on very simple syntactic rules and *ad hoc* heuristics. Since these methods are based on matching patterns rather than analyzing the program semantics, they do not lead to a generally applicable framework. Indeed, most of them handle only the simple case of *effectively thread-local* objects, i.e. locks that are declared as globally visible but are accessed only by one thread throughout the execution. For the many truly shared but still redundant locks, these existing methods are not effective.

We address this limitation by introducing a new and more generally applicable lock removal algorithm. Our method is generally applicable since it can remove not only the

[⋆] Chao Wang was supported in part by the NSF CAREER award CCF-1149454.

effectively thread-local locks but also the *truly shared* redundant locks. Our method is also efficient since it is based on a compositional analysis that involves only thread-local computation. Our method is sound in that it can guarantee preservation of the behavior of the original computation.

In formulating our lock removal strategy, we start from the classical notion of a concurrent computation as a happens-before relation on the shared variable accesses or, equivalently, as a set of partial orders. Two interleavings are equivalent if they induce the same partial order of shared variable accesses. Since removing locks lifts the corresponding mutual exclusion constraints, some previously infeasible thread interleavings may become feasible. Thus there is a danger for lock removal to introduce new program behaviors. To address this problem, we make sure that new interleavings are added by lock removal only if they do not add new partial orders. This leads to the formulation of the *behavior preservation* theorem, which is a main contribution of this paper.

Another main contribution is the set of *efficiently checkable* conditions under which the behavior preservation is guaranteed. They reduce the semantic check of behavior preservation to a simple static check of the feasibility of transitions between global control states. This is significant because it allows us to avoid enumerating the often astronomically large number of thread interleavings. Our method is thread-modular in that it does not require inspecting the interleaved parallel composition of threads. In addition, our focus on a concrete execution trace is also crucial in keeping the method scalable. The concrete execution trace provides the exact memory addresses that are accessed by each thread, thereby giving us the precise points-to information of lock pointers, together with information about the actual array fields accessed, etc.

Trace-based concurrent program analysis has obvious applications not only in runtime verification, but also in debugging, just-in-time (JIT) optimization, program understanding, and maintenance. An important feature of trace-based analysis is that the trace program has finitely many threads and a fixed set of named locks. Although the whole program may have pointers, loops, recursion, and dynamic thread creation, in the trace program, each thread is reduced to a bounded straight-line path. Most of the complications common to static program analysis are avoided because, during the concrete execution, branching decisions at if-else statements have been made, function calls have been inlined, loops have been unrolled, and recursions have been applied. The only remaining source of nondeterminism comes from thread interleaving.

We have implemented the proposed method in a runtime verification platform called *Fusion*, where the underlying bug detection algorithm uses an SMT-based symbolic analysis. Since redundant locks can introduce a large set of synchronization constraints during the modeling and checking phases, their presence often significantly increases the cost of the symbolic analysis. Our lock removal method has been used to remove these redundant locks. Our experiments on a set of public Java and C programs showed a significant reduction in the number of locks, which in turn led to a significant speedup in the subsequent symbolic analysis.

To sum up, this paper has made the following two contributions: (1) formulating the general framework of behavioral preservation to soundly remove redundant locks; and (2) proposing a set of efficiently checkable conditions based on the thread-local computation of lock access patterns.

The remainder of this paper is organized as follows. In Section 2, we use two examples to illustrate both the benefit and challenges of lock removal. In Section 3, we illustrate our main ideas. In Section 4, we present a set of efficiently checkable conditions. In Section 5, we demonstrate the application of our algorithm on the running example. Our experimental results are presented in Section 6. We review the related work in Section 7 and give our conclusions in Section 8.

2 Motivation

The main driving application in this paper is runtime predictive analysis [12, 25, 5, 11, 23, 29, 19], which is a promising method for detecting concurrency bugs by analyzing an execution trace. In other words, even if the given test execution is not erroneous, but if an alternative interleaving of the events of that trace can trigger a failure, runtime predictive analysis will be able to detect it. Since a concurrent program often has a very large number of sequential paths and thread interleavings, statically analyzing the whole program is often extremely difficult. In such cases, runtime predictive analysis offers a good compromise between runtime monitoring and full-fledged model checking.

Runtime predictive analysis typically has three steps: (1) run a test of the concurrent program to obtain an execution trace; (2) run a sound static analysis of the trace to compute all the *potential* violations, e.g. deadlocks and race conditions; (3) for each potential violation, build a precise predictive model to decide whether the violation is feasible. The main scalability bottleneck is step 3 wherein the feasibility check needs to explore all possible interleavings of the trace events. Although the problem in step 3 can be solved by an efficient symbolic analysis [29, 19], redundant locks in the trace program can unnecessarily increase the cost of this analysis, since they can lead to a large number of locking constraints that need to be modeled and checked. Our lock removal method can cut down on the number of unnecessary locking constraints, therefore resulting in significant performance improvement in the subsequent analysis.

```
T1() {               T2() {

0a: —-;              0b: —-;
1a: lock(A);         1b: a[10]++;
2a: a[1]++;          2b: a[11]++;
3a: unlock(A);       3b: lock(A);
4a: a[2]++;          4b: lock(B);
5a: lock(A);         5b: a[12]++;
6a: lock(B);         6b: unlock(B);
7a: a[3]++;          7b: unlock(A);
8a: unlock(B);       8b: sh++;
9a: unlock(A);
10a: sh++;

}                    }
```

(a) original

```
T1'() {              T2'() {

0a: —-;              0b: —-;
1a: lock(A);         1b: —-;
2a: —-;              2b: —-;
3a: unlock(A);       3b: lock(A);
4a: —-;              4b: lock(B);
5a: lock(A);         5b: —-;
6a: lock(B);         6b: unlock(B);
7a: —-;              7b: unlock(A);
8a: unlock(B);       8b: sh++;
9a: unlock(A);
10a: sh++;

}                    }
```

(b) intermediate

```
T1''() {             T2''() {

0a: —-;              0b: —-;
1a: —-;              1b: —-;
2a: —-;              2b: —-;
3a: —-;              3b: —-;
4a: —-;              4b: —-;
5a: —-;              5b: —-;
6a: —-;              6b: —-;
7a: —-;              7b: —-;
8a: —-;              8b: sh++;
9a: —-;
10a: sh++;

}                    }
```

(c) final

Fig. 1. Example: removing redundant lock statements from a concurrent trace program.

Consider the concurrent trace program in Fig. 1 (a), which has two straight-line paths in threads T_1 and T_2, respectively. The global variables are sh and array a. Suppose that the goal is to check whether locations $10a$ and $8b$ are simultaneously reachable (e.g. a data race), we need to decide whether there exists a valid interleaving of these trace statements along which T_1 and T_2 can reach $10a$ and $8b$, respectively.

First, note that precise knowledge of the memory accesses is available since the trace program is derived from a concrete execution. The knowledge can be used to cut down the number of shared accesses that need to be interleaved. For example, although $a[i]$ is a global variable, the entries of a accessed by the two threads in this particular trace program are all disjoint and can be treated as thread-local. In other words, we can use the runtime information to *slice away* the redundant statements. This can reduce the trace program in Fig. 1 (a) to the one in Fig. 1 (b).

Next, consider the program in Fig. 1 (b). Since locks A and B now protect only thread-local statements, some of these lock statements may be redundant. We shall show in later sections that, for this particular example, these lock statements are all redundant and therefore can be removed while preserving the original program behavior. This reduction yields the simple trace program shown in Fig. 1 (c) with only the shared variable accesses. Consequently, it becomes easy to decide the simultaneous reachability of $10a$ and $8b$.

Challenges in Lock Removal. The example in Fig. 1 may give a false impression that *locks protecting only thread-local operations can always be removed*. This is not true, as demonstrated by Fig. 2. In this example, variable sh=0 initially. The assertion at b_7 holds because, to get value 2, one has to execute $b_1 \ldots b_3 \rightarrow a_1 \ldots a_6 \rightarrow b_4 \ldots b_7$, which is impossible since lock A is held by thread T_2 at b_3, which prevents thread T_1 from acquiring the same lock at location a_2. However, if we remove the lock/unlock statements at a_2 and a_4 – since they protect only thread-local operations – the assertion at b_7 may fail because the aforementioned interleaving is now allowed. This example highlights the fact that locks may play a key role in defining the set of allowed program behaviors even if they do not guard any global operation. It also shows that, without a rigorous concurrency analysis, *ad hoc* heuristics are often susceptible to subtle errors. We address this problem by proposing a generally applicable lock removal framework.

```
T1() {              T2() {

a1 : sh ++;         b1 : ......
a2 : lock(A)        b2 : lock(A)
a3 : ......         b3 : sh=0;
a4 : unlock(A)      b4 : x=sh;
a5 : ......         b5 : unlock(A)
a6 : sh ++;         b6 : ......
a7 : ......         b7 : assert(x!=2);
}                   }
```

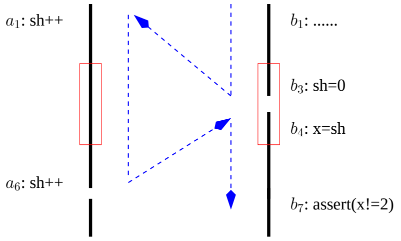


Fig. 2. Example: Assuming that sh=0 initially. The lock statements at a_2 and a_4 cannot be removed despite that they do not protect any shared access. Otherwise, assertion at b_7 may fail.

3 Lock Removal: the Core Idea

We say that a program $\mathcal{P}'$ results from another program $\mathcal{P}$ via lock removal if $\mathcal{P}'$ is obtained from $\mathcal{P}$ by converting some of the lock statements to `nop`. A lock statement in $\mathcal{P}$ is considered as redundant if removing that statement does not alter the program behavior. Here the program behavior is defined as the set of interleaved computations that are allowed by the program semantics. Since lock statements impose mutual exclusion constraints, they restrict the thread interactions. By removing lock statements from $\mathcal{P}$, in general, we may allow the new program $\mathcal{P}'$ to have more interleavings; on the other hand, it is impossible to remove any previously allowed interleavings in $\mathcal{P}$. Therefore, to preserve the program behavior, we only need to ensure that every newly added interleaving (allowed in $\mathcal{P}'$ but not in $\mathcal{P}$) is equivalent, in some sense, to an existing interleaving in $\mathcal{P}$. In other words, lock removal is sound as long as it does not add new equivalence classes (of interleavings).

3.1 The Lock Removal Strategy

Since characterizing interleavings directly is cumbersome and computationally expensive, we rely on the standard notion of concurrent computations as happens-before relations on the shared variable accesses [20, 14]. That is, executing two operations from different threads that update the same memory location in different orders may lead to different results. Therefore, instead of preserving interleavings of all the statements, we focus on preserving the partial orders of shared variable accesses (reads and writes).

For a program $\mathcal{P}$ comprised of the n threads $T_1, ..., T_n$, a global control state s is a tuple $(c_1, ..., c_n)$ where c_i is a control location of T_i for all $i \in [1..n]$. In contrast to a *concrete* program state, denoted $s \in \mathsf{s}$, the global control state s is more *abstract* in that it tracks only the program counters but not the values of the program variables. Therefore s can be viewed as a set of concrete states. Since thread-local operations are invisible to the other threads, in the sequel we shall assume without loss of generality that the locations in $(c_1, ..., c_n)$ are all starting points of **global operations**, i.e. either shared reads/writes or lock acquisitions. This restriction can drastically cut down the number of global control states that need to be considered during our analysis. Note that if a thread is at location c_i, it means that the operation at c_i has not been executed yet.

Definition 1 (Visible Successor). *For global control states s, s' in program $\mathcal{P}$, we say that s' is a visible successor of s iff there exist states $s \in \mathsf{s}$ and $s' \in \mathsf{s}'$ such that*

- *s' is reachable from s via a valid concurrent computation, and*
- *along this computation, the first operation is the only global operation.*

Our lock removal strategy can be phrased as follows: Removing all lock statements such that no new visible successor is introduced to any global control state that is reachable from the initial state in $\mathcal{P}$. In other words, for each s, if we can preserve the set of global control states that s can transit to, the program behavior will be preserved.

Consider Fig. 2 as an example. For all transitions between two global control locations, e.g. from (a_2, b_3) to (a_6, b_3), our lock removal strategy says that, if the transition is not allowed by $\mathcal{P}$ before lock removal, it should not be allowed by $\mathcal{P}'$ either. Based on this strategy, the lock statements at a_2 and a_4 will be preserved, because removing them would make the infeasible transition in $\mathcal{P}$ from (a_2, b_3) to (a_6, b_3) feasible in $\mathcal{P}'$.

3.2 Conservative Static Check

Although the lock removal strategy proposed so far is sound as well as generally applicable, computing the visible successors of a global control state is a challenging task, because the conditions in Definition 1 are semantic conditions. Checking the reachability between two concrete states s and s' would be too expensive in practice. To avoid this bottleneck, we introduce a set of checks based on the notion of *static* or *control-state* reachability.

Let $\mathsf{s} = (c_1, \ldots, c_n)$ and $\mathsf{s}' = (c'_1, ..., c'_n)$ be two global control states, where for each $i \in [1..n]$, the local path x^i of T_i leads from location c_i to c'_i. We say that s' is *statically* reachable from s if and only if there exists an interleaving of $x^1, ..., x^n$ that obeys the scheduling constraints imposed by the locks while ignoring data (which is the consistency between shared variable accesses).

Definition 2 (Static Visible Successor). *For global control states* s, s' *in program* $\mathcal{P}$*, we say that* s' *is a static visible successor of* s *iff*

- s *is statically reachable from* s *via some interleaved computation, and*
- *along this computation, at most one global operation is present.*

Here the second condition ensures that s' can be immediately reached from s (hence a successor). Let $\mathsf{Succ}_{\mathcal{P}}(\mathsf{s})$ be the set of static visible successors of s in program $\mathcal{P}$. Our static lock removal strategy is stated as follows.

Theorem 1 (Behavior Preservation). *Let program* $\mathcal{P}'$ *result from program* $\mathcal{P}$ *via lock removal. If for each global control state* s *of* $\mathcal{P}$*, we have* $\mathsf{Succ}_{\mathcal{P}}(\mathsf{s}) = \mathsf{Succ}_{\mathcal{P}'}(\mathsf{s})$*, then the two programs have the same behavior as defined by the partial orders of global operations.*

Intuitively, if no new global control state becomes reachable from the initial state, then there is certainly no new program behavior. For brevity, we omit the proof. A crucial property of Theorem 1 is that the static reachability check can be turned into a conceptual lock removal procedure as follows:

1. Enumerate the set S of global control states of the given trace program.
2. For each $\mathsf{s} \in \mathsf{S}$, compute the set $\mathsf{Succ}_{\mathcal{P}}(\mathsf{s})$of static visible successors.
3. For each lock statement *lk-stmt* in thread T_i, if there exists a global control location s such that, removing *lk-stmt* would add a new successor s' that is not in $\mathsf{Succ}_{\mathcal{P}}(\mathsf{s})$, we must retain *lk-stmt* else *lk-stmt* is removed.

There are two remaining problems. First, given two global control states s, s', how to efficiently decide whether s' is a static visible successor of s. Second, how to efficiently compute the set of static visible successors of s while avoiding the naive enumeration of all global control states. We will address these two problems in the next section.

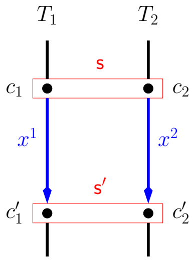
4 Compositional Lock Removal

We present a compositional analysis for static lock removal to avoid the exponential blowup incurred by naively enumerating the global control states. Our method is thread-modular in that the lock removal computation involves only thread-local reasoning, and therefore has a linear worst-case time complexity in the program size.

4.1 Deciding Static Reachability

We leverage an existing procedure [18] to decide the static reachability between two global control states. The procedure is both sound and complete for 2-threaded programs with nested locks. For programs with more than two threads, the procedure remains sound but is not complete. This is acceptable because, as long as it shows that s' is statically *unreachable* from s, the unreachability is guaranteed to hold.

The procedure in [18] can be viewed as a generalization of the standard *lockset* analysis [24]. The key insight is that, to decide whether $\mathsf{s}' = (c'_1, c'_2)$ is statically reachable from $\mathsf{s} = (c_1, c_2)$, for example, in a 2-threaded program, merely checking the disjointness of the set of locks held by T_1 and T_2 at c'_1 and c'_2 is not enough (see the figure on the right). Although overlapping locksets prove that s' is not reachable from s, the disjointness of the locksets is not sufficient to prove that s' is reachable from s. Instead, reachability can be decided more accurately by first computing a *lock access pattern (LAP)* for each path from c_i to c'_i, where $i \in [1..2]$, and then checking whether the LAPs are consistent.



Definition 3 (Lock Access Pattern). *The lock access pattern for path x^i from c_i to c'_i in thread T_i, denoted $\mathsf{LAP}(c_i, c'_i)$, is a tuple $(L_1, L_2, \mathsf{bah}, \mathsf{fah}, Held, Acq)$ where*

- *L_1 and L_2 are the set of locks held by T_i at c_i and c'_i, respectively;*
- *bah and fah are the backward and forward acquisition histories, respectively:*
  - *for each lock $l \in L_2$ held at c'_i, $\mathsf{bah}(l)$ is the set of locks acquired (and possibly released) after the last acquisition of l along path x^i from c_i to c'_i;*
  - *for each lock $l \in L_1$ held at c_i, $\mathsf{fah}(l)$ is the set of locks released (and possibly acquired) since the last release of l in traversing x^i backward from c'_i to c_i.*
- *$Held$ is the set of locks that are held in every state along path x^i from c_i to c'_i;*
- *Acq is the set of locks that are acquired (and possibly released) along path x^i.*

A key feature of this LAP-based static analysis procedure is that all computations are local to each individual thread, which is crucial in ensuring scalability.

Decomposition Result. The static reachability from s to s' can be decided by checking whether the corresponding lock access patterns are consistent. For ease of exposition, we present the result for programs with two threads. However, the result, as well as all the other subsequent results, is applicable to programs with n threads.

Let $\mathsf{s} = (c_1, c_2)$ and $\mathsf{s}' = (c'_1, c'_2)$ be two global control states, and $\mathsf{LAP}(c_1, c'_1) = (L^1_1, L^1_2, \mathsf{bah}^1, \mathsf{fah}^1, Held^1, Acq^1)$ and $\mathsf{LAP}(c_2, c'_2) = (L^2_1, L^2_2, \mathsf{bah}^2, \mathsf{fah}^2, Held^2, Acq^2)$ be the lock access patterns. Then s' is statically reachable from s iff

1. $L_1^1 \cap L_1^2 = \emptyset$, and $L_2^1 \cap L_2^2 = \emptyset$;
2. there do not exist locks $l \in L_1^1$ and $l' \in L_1^2$ such that $l \in \mathsf{fah}^2(l')$ and $l' \in \mathsf{fah}^1(l)$;
3. there do not exist locks $l \in L_2^1$ and $l' \in L_2^2$ such that $l \in \mathsf{bah}^2(l')$ and $l' \in \mathsf{bah}^1(l)$;
4. $Acq^1 \cap Held^2 = \emptyset$, and $Acq^2 \cap Held^1 = \emptyset$.

For n-threaded programs, the only significant difference would be in conditions 2 and 3, wherein one has to account for the cases in which n threads form a cyclic dependency that may span multiple threads instead of just two.

4.2 Compositional Analysis

To avoid the expensive enumeration of global control states as described in Theorem 1, we compute for each individual thread, all pairs of local control states that may corresponds to some static visible successors. More specifically, a pair (c_i, c_i') of control locations in thread T_i is called a *pair of interest (POI)* iff

- c_i and c_i' correspond to either shared variable accesses or lock acquisitions, and
- there exists a local path x^i in T_i from c_i to c_i' such that no other shared variable access or lock acquisition occurs between c_i and c_i'.

Our compositional lock removal procedure is given in Algorithm 1. After computing the POIs of each thread T_i, it traverses that thread to collect the lock access patterns for all POIs. Let LP_i denote the set of all lock access patterns in T_i. Note that LP_i can be computed via a single traversal pass of thread T_i (step 4).

Algorithm 1 Compositional Lock Removal

```
 1: Input: Threads $T_1, T_2$
 2: for each thread $T_i$ do
 3:     Enumerate all pairs of interest $\mathsf{POI}(T_i)$.
 4:     Traverse the local path in $T_i$ to compute $\mathsf{LAP}(c_i, c_i')$ for each pair $(c_i, c_i') \in \mathsf{POI}(T_i)$.
 5:     Let $\mathsf{LP}_i$ be the set of lock access patterns of all POIs in $T_i$.
 6: end for
 7: for each pair $(lap_1, lap_2)$ where $lap_i \in \mathsf{LP}_i$ for all thread index $i \in [1..2]$ do
 8:     if $lap_1, lap_2$ are inconsistent then
 9:         Identify the set of lock statements that are the root causes of inconsistency.
10:     end if
11: end for
12: Remove lock statements that are not the root causes of inconsistency for any pair.
```

Instead of iterating through the set of all global control states, Algorithm 1 considers all pairs (lap_1, lap_2) of lock access patterns that are inconsistent (step 7). Note that lap_i corresponds to some pair $(c_i, c_i') \in \mathsf{POI}(T_i)$ and the inconsistency of lap_1 and lap_2 means that there exist some lock statements that prevent (c_1, c_2) from reaching (c_1', c_2'). In this case, we need to identify a minimum subset of lock statements that are sufficient to establish this inconsistency, and retain these lock statements. Finally any lock statement that is not responsible for causing an inconsistency between any pair of lock access patterns does not impact the reachability between any pair of global control states, and is therefore removed.

It is worth pointing out that the lock statements (to be retained) can be identified from the lock access patterns (lap_1 and lap_2) alone, without considering the global control states or the POIs that generate these lock access patterns. In other words, we can implicitly isolate the set of non-reachable pairs of global control states without explicitly enumerating them. The algorithm can also be extended to programs with n threads, by changing step 7 to check for inconsistent tuples of the form $(lap_1, ..., lap_n)$, as opposed to the inconsistent pair (lap_1, lap_2).

4.3 Identifying the Locks to be Retained

If s' is not statically reachable from s in the original program $\mathcal{P}$, according to Section 4.1, at least one of the conditions in the decomposition result must be violated. From these conditions, we can isolate the root causes that prevent s from reaching s' statically. Our observation is that if s' is not statically reachable from s in $\mathcal{P}$, then we need to make sure that s' is not reachable from s in the transformed program $\mathcal{P}'$. The behavior preservation can be guaranteed if we retain at least some (but not all) of the lock statements that prevent s from reaching s'.

Given an inconsistent pair lap_1 and lap_2 of lock access patterns, we can define a reachability barrier by isolating the locks causing the inconsistency. To this end, for each pair $(\mathsf{s}, \mathsf{s}')$ of global control states where $\mathsf{s} = (c_1, c_2)$ and $\mathsf{s}' = (c'_1, c'_2)$, we define a *reachability barrier*, denoted $\mathsf{RB}(\mathsf{s}, \mathsf{s}')$, which is the set of all locksets (L) for which at least one of the following conditions holds:

- $L = \{l\}$, where l is held at both c_1 and c_2 or at both c'_1 and c'_2 (violating condition 1 of the decomposition result);
- $L = \{l, l'\}$, where l and l' are held at c_1 and c_2, respectively, such that $l \in \mathsf{fah}(l')$ and $l' \in \mathsf{fah}(l)$ (violation of condition 2);
- $L = \{l, l'\}$, where l and l' are held at c'_1 and c'_2, respectively, such that $l \in \mathsf{bah}(l')$ and $l' \in \mathsf{bah}(l)$ (violation of condition 3);
- $L = \{l\}$, where l is held throughout x^1 (or x^2) and is acquired along x^2 (or x^1) (violation of condition 4).

Note that in order to ensure that s' remains unreachable from s, it suffices to retain the locks belonging to some lockset in $\mathsf{RB}(\mathsf{s}, \mathsf{s}')$ as that will ensure that at least one condition of the decomposition result is violated.

5 Applying Lock Removal to the Running Example

We now use our new method to remove all locks in the trace program shown in Fig. 1 (b) while preserving the program behavior.

We start by identifying the pairs of interest. In the path x^1 shown in Fig. 1 (b), there are three lock acquisition statements, i.e. locations $1a$, $5a$ and $6a$, and two shared variable accesses, i.e., $0a$ and $10a$ (the initial state is always treated as a shared variable access). This leads to the pairs of interest $\mathsf{POI}(x^1) = \{(0a, 0a), (0a, 1a), (1a, 1a),$

$(1a, 5a)$, $(5a, 5a)$, $(5a, 6a)$, $(6a, 6a)$, $(6a, 10a)\}$. Similarly, $\mathsf{POI}(x^2) = \{(0b, 0b)$, $(0b, 3b)$, $(3b, 3b)$, $(3b, 4b)$, $(4b, 4b)$, $(4b, 8b)$, $(8b, 8b)\}$.

Next, we compute the lock access patterns generated by all pairs of interest in paths x^1 and x^2. Toward that end, we compute the lap2POI function for x^2 that maps each lock access pattern *lap* that is encountered to the set of POIs of x^2 that generate that pattern. For each (c_2, c_2') in the set $\{(0b, 0b)$, $(0b, 3b)$, $(3b, 3b)$, $(8b, 8b)\}$, no lock is held at either c_2 or c_2' and no lock is acquired along the sub-sequence of x^2 from c_2 to c_2'. Thus all the entries in the lock access pattern tuples for these pairs are empty (note that if a thread is at location $3b$ it means that the statement at $3b$ hasn't been executed yet, i.e., lock held at location $3b$ is $\emptyset$).

Consider now the pair of interest $(4b, 8b)$. We show that $\mathsf{LAP}(4b, 8b) = (\{A\}, \emptyset, \{(A, \{B\})\}, \emptyset, \emptyset, \{B\})$. The first two entries in the tuple are the locksets held at $4b$ and $8b$ which are $\{A\}$ and $\emptyset$, respectively. Since no lock is held at the final state $8b$, the forward acquisition histories, i.e., the fourth entry of the tuple is empty. On the other hand, lock A is held at the initial state $4b$. This lock is released at $7b$. However before it is released T_2 also releases B at $6b$. Thus B is in the backward acquisition history of A which is reflect in the third entry of the tuple. Also, since lock B is acquired at location $4b$, we have $Acq = \{B\}$ (6th entry). Finally, since there exists no lock that is held at all states, we have $Held = \emptyset$ (5th entry). Similarly, we may compute the lock access patterns for the remaining pairs of interest (see Fig. 3 (b)). Similarly, we compute the lap2POI function for x^1 (see Fig. 3 (a)).

From Fig. 3 (a) and 3 (b), we compute the inconsistent pairs (p_1, p_2) of lock access patterns where

1. $p_1 = (\{A\}, \{A\}, \emptyset, \emptyset, \{A\}, \emptyset)$, $p_2 = (\emptyset, \{A\}, \emptyset, \{(A, \{\})\}, \emptyset, \{A\})$: $Held$ and Acq fields of p_1 and p_2, respectively, have the common lock A.
2. $p_1 = (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \{A\})$ and $p_2 = (\{A\}, \{A\}, \emptyset, \emptyset, \{A\}, \emptyset)$: Acq and $Held$ fields of p_1 and p_2, respectively, have the common lock A.
3. $p_1 = (\emptyset, \{A\}, \emptyset, \{(A, \{\})\}, \emptyset, \{A\})$ and $p_2 = (\{A\}, \{A\}, \emptyset, \emptyset, \{A\}, \emptyset)$: Acq and $Held$ fields of p_1 and p_2, respectively, have the common lock A
4. $p_1 = (\{A\}, \{A\}, \emptyset, \emptyset, \{A\}, \emptyset)$ and $p_2 = (\{A\}, \{A\}, \emptyset, \emptyset, \{A\}, \emptyset)$: L_1 fields have the common lock A
5. $p_1 = (\{A\}, \{A\}, \emptyset, \emptyset, \{A\}, \emptyset)$ and $p_2 = (\{A\}, \emptyset, \{(A, \{B\})\}, \emptyset, \emptyset, \{B\})$: L_1 fields have the common lock A
6. $p_1 = (\{A\}, \emptyset, \{(A, \{B\})\}, \emptyset, \emptyset, \{B\})$ and $p_2 = (\{A\}, \{A\}, \emptyset, \emptyset, \{A\}, \emptyset)$: L_1 fields have the common lock A
7. $p_1 = (\{A\}, \emptyset, \{(A, \{B\})\}, \emptyset, \emptyset, \{B\})$ and $p_2 = (\{A\}, \emptyset, \{(A, \{B\})\}, \emptyset, \emptyset, \{B\})$: L_1 fields have the common lock A
8. $p_1 = (\emptyset, \{A\}, \emptyset, \{(A, \{\})\}, \emptyset, \{A\})$ and $p_2 = (\emptyset, \{A\}, \emptyset, \{(A, \{\})\}, \emptyset, \{A\})$: L_2 fields have the common lock A.

Note that in each of the above cases, the only lock occurring in the reachability barriers of the non-reachable pairs of global control states is A. Since lock B does not occur in any of the reachability barriers, in the first iteration, we can remove all statements locking/unlocking B.

Now we repeat the lock removal procedure again on the trace program in Fig. 1 (b), by converting statements $6a$, $8a$, $4b$ and $6b$ to `nop`. These new traces generate the lap2POI functions shown in Figs. 3 (c) and (d). Note that now all pairs of access patterns are mutually consistent. Thus the reachability barriers for all pairs of global control states are empty. Hence all locks in the original traces can now be removed giving us the traces with no lock statements.

$$\begin{array}{l}
(\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset) \rightarrow \{(0a,0a),(0a,1a),(1a,1a),(5a,5a),(10a,10a)\}\\
(\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\{A\}) \rightarrow \{(1a,5a)\}\\
(\emptyset,\{A\},\emptyset,\{(A,\{\})\},\emptyset,\{A\}) \rightarrow \{(5a,6a)\}\\
(\{A\},\{A\},\emptyset,\emptyset,\{A\},\emptyset) \rightarrow \{(6a,6a)\}\\
(\{A\},\emptyset,\{(A,\{B\})\},\emptyset,\emptyset,\{B\}) \rightarrow \{(6a,10a)\}
\end{array}$$

(a)

$$\begin{array}{l}
(\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset) \rightarrow \{(0b,0b),(0b,3b),(3b,3b),(8b,8b)\}\\
(\emptyset,\{A\},\emptyset,\{(A,\{\})\},\emptyset,\{A\}) \rightarrow \{(3b,4b)\}\\
(\{A\},\{A\},\emptyset,\emptyset,\{A\},\emptyset) \rightarrow \{(4b,4b)\}\\
(\{A\},\emptyset,\{(A,\{B\})\},\emptyset,\emptyset,\{B\}) \rightarrow \{(4b,8b)\}
\end{array}$$

(b)

$$\begin{array}{l}
(\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset) \rightarrow \{(0a,0a),(0a,1a),(1a,1a),(5a,5a),(10a,10a)\}\\
(\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\{A\}) \rightarrow \{(1a,5a),(5a,10a)\}
\end{array}$$

(c)

$$\begin{array}{l}
(\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset) \rightarrow \{(0b,0b),(0b,3b),(3b,3b),(8b,8b)\}\\
(\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\{A\}) \rightarrow \{(3b,8b)\}
\end{array}$$

(d)

Fig. 3. The lap2POI function for x^1 (left) and x^2 (right)

Generalizations. So far, for ease of exposition, we have presented all the algorithms using concurrent trace programs with two threads. However, our results can be extended to programs with an arbitrary but fixed number of threads. This generalizations do not require additional insights. The only difference from the 2-thread case is that we need an efficient technique to decide static reachability between global control states which are now n-tuples of the form $(c_1, ...c_n)$, where each c_i is either a shared variable access or a lock acquisition in thread T_i. This is achieved via a straightforward extension of the decomposition result in Section 4.1. That is, for each pair of threads, we check whether their lock access patterns (LAPs) are consistent.

So far we have discussed only mutex locks. A typical real-world concurrent program in Java or C (with POSIX threads) may have additional concurrency primitives such as thread creation and join operations, wait/notify/notifyall, as well as reentrant locks. The presence of these synchronization primitives does not affect the soundness of our lock removal algorithm. The reason is that, if s' is statically unreachable from s according to locks (while ignoring data and other concurrency primitives), it is guaranteed to be unreachable when more synchronization constraints are considered. At the same time, if there is a way to incorporate the causality constraints imposed by other concurrency primitives, one can more accurately determine the reachability between global control states, therefore leading to the identification and removal of potentially more redundant lock statements. To this end, we have incorporated the universal causality graph based analysis in [19] during our implementation of the proposed lock removal method. However, we note that this UCG-based analysis is orthogonal to lock removal, and can be carried out once in the beginning of the computation.

6 Experiments

We have evaluated the lock removal method in the context of an SMT-based runtime predictive analysis [28, 29], to quickly remove the lock statements that are redundant and therefore ease the burden of modeling and checking by the SMT solvers.

We now provide a brief overview of the symbolic predictive analysis. Given a multi-threaded Java or C program and a user-defined test case, the predictive analysis procedure first instruments the program code to add self-logging capability, and then uses stress tests to detect concurrency failures. However, due to the scheduling nondeterminism and the astronomically large number of interleavings, it is often difficult to uncover the concurrency bugs. If testing fails to detect any bug, we start a post-mortem analysis of the logged execution trace.

In this subsequent analysis, first we use a simple control flow analysis to compute the *potential bugs*. Consider the one-variable three-access atomicity violation [21, 11] as an example. In this case, a potential bug is a sequence $t_c ... t_r ... t_{c'}$ of program statements such that: (1) t_c and $t_{c'}$ are intended to be executed atomically by one thread, (2) t_r is in another thread and is data dependent with both t_c and $t_{c'}$. Then we use a more precise static analysis based on the *universal causality graph (UCG [19])* to prune away the obviously bogus violations.

For each remaining potential violation, we call the SMT-based symbolic procedure to decide if there exists a valid interleaving under which the violation is feasible. In this context, an interleaving is feasible if it satisfies both the synchronization consistency (e.g. locks) and the shared memory consistency. Please refer to [28, 29, 26] for more information about the symbolic encoding. Here we assume the sequential consistency (SC) memory model. We have used the YICES solver from SRI [8] in our experiments. Since having more lock statements generally leads to more logical constraints and therefore a higher cost for SMT solving, we have used lock removal before the SMT-based analysis, to remove the redundant lock statements.

We conducted experiments using the following benchmarks[1]. The Java programs come from various public benchmarks [16, 17, 15, 27]. The C programs are the PThreads implementation of two sets of known bug patterns. The first set (*At*) mimics an atomicity violation in the Apache web server code (c.f. [21]), where *At1* is the original program, while *At1a* and *At2a* are generated by adding code to the original programs to remove the atomicity violations. The second set (*bank*) is a parameterized version of the *bank* example [10], where the original program *bank-av* has a well-known atomicity violation and the remaining two are various attempts of fixing it All our experiments were conducted on a PC with 1.6 GHz Intel processor and 2GB memory running Fedora.

Table 1 shows the results. The first five columns show the statistics of the trace program, including the name, the number of threads, the total number of events, the number of lock/unlock events, and the number of named locks. The next nine columns show the statistics of the lock removal computation. In particular, Columns 6-9 show the total number of pairs of interest (POI), the number of POIs without any held lock (POI-e), the number of POIs with non-trivial lock acquisition histories (POI-h), and the maximum nesting depths of locks (max-h). The fact that *max-h* is often zero helps to

[1] The benchmarks are available at http://www.nec-labs.com/∼chaowang/pubDOC/LnW.tar.gz

make our analysis scale to real-life programs. Columns 10-11 show the total number of relevant pairs of global control states, and the number of pairs wherein one state is unreachable from the other. Columns 12 and 13 show the number of critical sections (pairs of lock-unlock statements) in the original and transformed programs, respectively. Column 14 shows the total time (in seconds) taken for the lock removal computation.

Table 1. Results: Using lock removal to improve symbolic analysis. *mem* means memory-out.

Concurrent Trace Program					Lock Removal Computation									Symbolic Analysis			
name	thrds	events	lk-evs	lk-v	POI	POI-e	POI-h	max-h	vis-ne	vis-ch	lk-r	rm-r	time(s)	p-avs	r-avs	pre(s)	post(s)
ra.Main	3	55	12	3	23	7	0	0	65	0	5	3	0.0	2	0	0.0	0.0
connect	4	97	16	1	43	29	0	0	1526	0	8	0	0.0	6	0	0.1	0.1
hedcex	1	122	35	7	1	0	0	0	0	0	0	0	0.0	0	0	0.0	0.0
liveness	7	283	44	9	105	68	0	0	10272	0	15	0	0.2	36	0	0.4	0.4
BarrierB1	10	653	108	2	307	168	0	0	69498	0	35	14	0.9	102	0	10.5	3.0
BarrierB2	13	805	136	2	409	217	0	0	120659	0	49	21	1.6	87	0	54.5	7.4
account1	11	902	146	21	230	134	0	0	43690	0	72	30	0.7	140	2	1.8	0.9
philo	6	1141	126	6	433	260	0	0	147294	0	63	10	2.2	81	0	42.5	19.4
account2	21	1747	282	41	442	260	0	0	171400	0	140	60	2.6	280	3	8.7	2.4
Daisy1	3	2998	422	10	843	105	29	1	17249	141	204	175	0.3	7	0	mem	21.3
Elevator1	4	3004	370	11	893	28	0	0	1453	0	184	174	0.1	4	0	29.6	0.7
Elevator2	4	5001	610	11	1992	116	0	0	25435	0	304	257	0.7	8	0	mem	4.3
Elevator3	4	8004	1128	11	2369	214	0	0	81890	0	563	468	1.9	12	0	mem	28.2
Tsp	4	45653	20	5	87	4	0	0	20	0	8	6	0.0	0	0	0.0	0.0
At1	3	88	6	1	14	7	0	0	60	0	3	0	0.0	3	0	1.0	0.0
At1a	3	100	8	1	17	10	0	0	126	0	4	0	0.0	4	0	1.0	0.0
At2a	3	462	126	2	156	149	32	1	38208	9216	52	16	0.6	52	16	2.0	0.6
Bank-av	3	748	20	3	160	104	0	0	28776	0	40	8	0.4	40	8	8.0	0.4
Bank-sav	3	852	28	3	195	139	0	0	51510	0	56	8	0.7	56	8	8.0	0.7
Bank-fix	3	856	32	3	204	147	16	1	57612	12540	64	8	0.8	64	8	9.0	0.8

Finally, the last four columns in Table 1 show the impact of lock removal on the performance of a runtime verification procedure. Recall that, for each of the potential atomicity violations, we use symbolic analysis to decide whether it is a real atomicity violation. Here we first show the total number of potential atomicity violations (p-avs) that are collected by a simple static analysis, and then show the number of real atomicity violations found by the precise symbolic analysis (r-avs). Please refer to [29, 19] for more details on predicting atomicity violation. The last two columns compare the runtime of symbolic analysis with and without lock removal. The results clearly show that lock removal has made the predictive verification step more efficient. Note that for *Daisy1* (which is file system) and *Elevator2*, without lock removal, symbolic execution would run out of the 2GB memory limit, whereas after lock removal, they were able to finish in short time.

7 Related Work

Existing work on automatically removing unnecessary synchronizations has concentrated mostly on performance optimization and on eliminating thread-local locks [3, 4, 6, 30], i.e. locks that have been acquired or released by a single thread or used to protect an object accessed by a single thread. The difference among these methods lies in how they identify shared/escaped objects. For example, Blanchet [3] uses a flow-insensitive escape analysis both to allocate thread-local objects on the stack and to eliminate synchronization from stack-allocated objects. Bogda et. al. [4] also use a flow-insensitive escape analysis to eliminate synchronization from thread local objects, but the analysis is limited to thread-local objects that are only reachable by paths of one or two references from the stack. Choi et al. [6] perform an inter-procedural points-to analysis to classify objects as globally escaping, escaping via an argument, and not escaping. When synchronizing, the compiler eliminates synchronizations for thread-local objects, while preserving Java semantics by flushing the local processor cache.

Ruf [22] combines a thread behavior analysis with a unification based alias analysis to removal unnecessary synchronizations. Aldrich et al. [1] propose three analysis to optimize the synchronization opportunities: lock analysis, unshared field analysis, and multithreaded object analysis. Lock analysis computes a description of the monitors held at each synchronization point so that reentrant locks and enclosed locks can be eliminated. Unshared field analysis identifies unshared fields so that lock analysis can safely identify enclosed locks. Finally, multithreaded object analysis identifies which objects may be accessible by more than one thread. This enables the elimination of all synchronization on objects that are not multi-threaded. Zee and Rinard [30] present a static program analysis for removing unnecessary write barriers in Java programs that use generational garbage collection.

In contrast, the focus of our work is not to identify which objects are *effectively thread-local*, which objects are shared, or when they are shared, by multiple threads, but to identify more optimization opportunities on the truly shared objects and yet redundant locks. To the best of our knowledge, this is the first such lock removal algorithm. It is generally applicable, based on a rigorous and unified concurrency analysis framework. It is also practically efficient, due to the use of lock access patterns, which involves only thread-local computation.

In the formulation of our efficient check for behavior preservation, we have leveraged the lock access patterns [18], since our trace program has a fixed number of threads interacting with only nested locks. To extend the method from trace programs to whole programs, one might need to leverage the more advanced machinery in [13, 9] to deal with locks interacting with dynamic thread creation.

In the literature, there has also been some work on reducing the run-time cost of synchronizations, e.g. by making their implementation more efficient (e.g. [2]) rather than removing the unnecessary ones. These techniques complement ours. Our local removal algorithm is also different from lock coarsening [7], which optimizes the necessary synchronizations, e.g. those arising from acquiring and releasing a lock multiple times in succession. Converting multiple lock operations into one, in general, changes the program behavior, and therefore one must take care not to introduce deadlock.

8 Conclusions

In this paper, we have presented an efficient and fully automatic lock removal technique for concurrent trace programs. A key feature of our method is that it is compositional in nature, i.e., hinges on a thread local analysis, which makes it applicable to large, realistic programs. Furthermore, our technique guarantees the preservation of program behaviors, i.e., partial orders induced on shared variable accesses. These features make it a standalone utility with many wide ranging applications, including performance optimization as well as improving the efficacy of concurrent program analysis like runtime verification, model checking and dataflow analysis. As a concrete application, we demonstrated the use of our lock removal technique in enhancing the scalability of predictive analysis in the context of runtime verification of concurrent programs.

References

1. J. Aldrich, C. Chambers, E. G. Sirer, and S. J. Eggers. Static analyses for eliminating unnecessary synchronization from Java programs. In *International Symposium on Static Analysis*, pages 19–38, 1999.
2. D. F. Bacon, R. B. Konuru, C. Murthy, and M. J. Serrano. Thin Locks: Featherweight synchronization for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–268, 1998.
3. B. Blanchet. Escape analysis for object-oriented languages: Application to Java. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 20–34, 1999.
4. J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 35–46, 1999.
5. F. Chen and G. Rosu. Parametric and sliced causality. In *International Conference on Computer Aided Verification*, pages 240–253. Springer, 2007.
6. J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Trans. Program. Lang. Syst.*, 25(6):876–910, 2003.
7. P. C. Diniz and M. C. Rinard. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. *J. Parallel Distrib. Comput.*, 49(2):218–244, 1998.
8. B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In *International Conference on Computer Aided Verification*, pages 81–94. Springer, 2006. LNCS 4144.
9. J. Esparza and P. Ganty. Complexity of pattern-based verification for multithreaded programs. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 499–510, 2011.
10. E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *Parallel and Distributed Processing Symposium*, page 286, 2003.
11. A. Farzan and P. Madhusudan. Meta-analysis for atomicity violations under nested locking. In *International Conference on Computer Aided Verification*, pages 248–262, 2009.
12. C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Parallel and Distributed Processing Symposium*, 2004.
13. T. M. Gawlitza, P. Lammich, M. Müller-Olm, H. Seidl, and A. Wenner. Join-lock-sensitive forward reachability analysis for concurrent programs with dynamic process creation. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 199–213, 2011.
14. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*. Springer, 1996.
15. K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Software Tools for Technology Transfer*, 2(4), 2000.

16. http://research.microsoft.com/qadeer/cav issta.htm. Joint cav/issta special even on specification, verification, and testing of concurrent software.
17. http://www2.epcc.ed.ac.uk/computing/ research_activities/java_grande/index_1.html. The java grande forum benchmark suite.
18. V. Kahlon. Boundedness vs. unboundedness of lock chains: Characterizing decidability of pairwise cfl-reachability for threads communicating via locks. In *Symposium on Logic in Computer Science*, pages 27–36, 2009.
19. V. Kahlon and C. Wang. Universal Causality Graphs: A precise happens-before model for detecting bugs in concurrent programs. In *International Conference on Computer Aided Verification*, pages 434–449, 2010.
20. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.
21. S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Architectural Support for Programming Languages and Operating Systems*, pages 37–48, 2006.
22. E. Ruf. Effective synchronization removal for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 208–218, 2000.
23. C. Sadowski, S. N. Freund, and C. Flanagan. Singletrack: A dynamic determinism checker for multithreaded programs. In *European Symposium on Programming*, pages 394–409, 2009.
24. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.
25. K. Sen, G. Rosu, and G. Agha. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In *Formal Methods for Open Object-Based Distributed Systems*, pages 211–226, 2005.
26. N. Sinha and C. Wang. On interference abstractions. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 423–434, 2011.
27. C. von Praun and T. R. Gross. Static detection of atomicity violations in object-oriented programs. *Object Technology*, 3(6), 2004.
28. C. Wang, S. Kundu, M. Ganai, and A. Gupta. Symbolic predictive analysis for concurrent programs. In *International Symposium on Formal Methods*, pages 256–272, 2009.
29. C. Wang, R. Limaye, M. Ganai, and A. Gupta. Trace-based symbolic analysis for atomicity violations. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 328–342, 2010.
30. K. Zee and M. C. Rinard. Write barrier removal by static analysis. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 191–210, 2002.