Reasoning About Threads Communicating via Locks

Vineet Kahlon, Franjo Ivančić, and Aarti Gupta

NEC Labs America, 4 Independence Way, Suite 200,
Princeton, NJ 08536, USA
{kahlon, ivancic, agupta}@nec-labs.com

Abstract. We propose a new technique for the static analysis of concurrent programs comprised of multiple threads. In general, the problem is known to be undecidable even for programs with only two threads but where the threads communicate using CCS-style pairwise rendezvous [11]. However, in practice, a large fraction of concurrent programs can either be directly modeled as threads communicating solely using locks or can be reduced to such systems either by applying standard abstract interpretation techniques or by exploiting separation of control from data. For such a framework, we show that for the commonly occurring case of threads with nested access to locks, the problem is efficiently decidable. Our technique involves reducing the analysis of a concurrent program with multiple threads to individually analyzing augmented versions of the given threads. This not only yields decidability but also avoids construction of the state space of the concurrent program at hand and thus bypasses the state explosion problem making our technique scalable. We go on to show that for programs with threads that have non-nested access to locks, the static analysis problem for programs with even two threads becomes undecidable even for reachability, thus sharpening the result of [11]. As a case study, we consider the Daisy file system [1] which is a benchmark for analyzing the efficacy of different methodologies for debugging concurrent programs and provide results for the detection of several bugs.

1 Introduction

Multi-threading is a standard way of enhancing performance by exploiting parallelism among the different components of a computer system. As a result the use of concurrent multi-threaded programs is becoming pervasive. Examples include operating systems, databases and embedded systems. This necessitates the development of new methodologies to debug such systems especially since existing techniques for debugging sequential programs are inadequate in handling concurrent programs. The key reason for that is the presence of many possible interleavings among the local operations of individual threads giving rise to subtle unintended behaviors. This makes multi-threaded software behaviorally complex and hard to analyze thus requiring the use of formal methods for their validation.

One of the most widely used techniques in the validation of sequential programs is dataflow analysis [12] which can essentially be looked upon as a combination of abstract interpretation and model checking [13]. Here, abstract interpretation is used to get a finite representation of the control part of the program while recursion is modeled using a stack. Pushdown systems (PDSs) provide a natural framework to model such

K. Etessami and S.K. Rajamani (Eds.): CAV 2005, LNCS 3576, pp. 505–518, 2005.

abstractly interpreted structures. A PDS has a finite control part corresponding to the valuation of the variables of the program and a stack which provides a means to model recursion. Dataflow analysis then exploits the fact that the model checking problem for PDSs is decidable for very expressive classes of properties - both linear and branching time (cf. [2, 16]).

Following data-flow analysis for sequential programs, we model a multi-threaded program as a system comprised of multiple pushdown systems interacting with each other using a communication mechanism like a shared variable or a synchronization primitive[1]. While for a single PDS the model checking problem is efficiently decidable for very expressive logics, it was shown in [11] that even simple properties like reachability become undecidable even for systems with only two threads but where the threads communicate using CCS-style pairwise rendezvous.

However, in a large fraction of real-world concurrent software used, for example, in file systems, databases or device drivers, the key issue is to resolve conflicts between different threads competing for access to shared resources. Conflicts are typically resolved using locks which allow mutually exclusive access to a shared resource. Before a thread can have access to a shared resource it has to acquire the lock associated with that resource which is released after executing all the intended operations. For such software, the interaction between concurrently executing threads is very limited making them loosely coupled. For instance, in a standard file system the control flow in the implementation of the various file operations is usually independent of the data being written to or read from a file. Consequently such programs can either be directly modeled as systems comprised of PDSs communicating via locks or can be reduced to such systems either by applying standard abstract interpretation techniques or by exploiting separation of control and data. Therefore, in this paper, we consider the model checking problem for PDSs interacting using locks.

Absence of conflicts and deadlock freedom are among the most crucial properties that need to be checked for multi-threaded programs, particularly because checking for these is usually a pre-cursor for verifying more complex properties. Typical conflicts include, for example, data races where two or more threads try to access a shared memory location with at least one of the accesses being a write operation. This and most other commonly occurring conflicts (can be formulated to) occur pairwise among threads. With this in mind, given a concurrent program comprised of the n threads T_1,...,T_n, we consider correctness properties of the following forms:

- Liveness (Single-indexed Properties): $\mathsf{E}h(i)$ and $\mathsf{A}h(i)$, where $h(i)$ is an LTL\X formula (built using F "eventually," U "until," G "always," but without X "next-time") interpreted over the local control states of the PDS representing thread T_i, and E (for some computation starting at the initial global configuration) and A (for all computations starting at the initial global configuration) are the usual path quantifiers.
- Safety (Double-indexed Properties): $\bigwedge_{i \neq j} \mathsf{EF}(a_i \wedge b_j)$, where a_i and b_j are local control states of PDSs T_i and T_j, respectively.
- Deadlock Freedom.

[1] Henceforth we shall use the terms thread and PDS interchangeably.

For single-indexed properties, we show that the model checking problem is efficiently decidable. Towards that end, given a correctness property $\mathsf{E}h(i)$ over the local states of thread T_i, we show how to reduce reasoning in an exact, i.e., sound and complete, fashion about a system with n threads to a system comprised of just the thread T_i. This reduces the problem of model checking a single-indexed LTL$\backslash$X formula for a system with n threads to model checking a single thread (PDS), which by [2] is known to be efficiently decidable.

The model checking problem for double-indexed properties is more interesting. As for single-indexed properties, we show that we can reduce the model checking problem for $\mathsf{E}h(i,j)$ for a system with n threads to the system comprised of just the two threads T_i and T_j. However, unlike the single index case, this still does not yield decidability of the associated model checking problem. We show that, in general, the problem of model checking remains undecidable even for pairwise reachability, viz., properties of the form $\mathsf{EF}(a_i \wedge b_j)$, where a_i and b_j are local control states of thread T_i and T_j, even for programs with only two threads.

However, most real-world concurrent programs use locks in a nested fashion, viz., each thread can only release the lock that it acquired last and that has not yet been released. Indeed, practical programming guidelines used by software developers often require that locks be used in a nested fashion. In fact, in Java and C# locking is syntactically guaranteed to be nested. In this case, we show that we can reduce reasoning about pairwise reachability of a given two-threaded program to individually model checking augmented versions of each of the threads, which by [2] is efficiently decidable. The augmentation involves storing for each control location of a thread the history of locks that were acquired or released in order to get to that location. We show that storing this history information guarantees a sound and complete reduction. Furthermore, it avoids construction of the state space of the system at hand thereby bypassing the state explosion problem thus making our technique scalable to large programs. Thus we have given an efficient technique for reasoning about threads communicating via locks which can be combined synergistically with existing methodologies.

As a case study, we have applied our technique to check race conditions in the Daisy file system [1] and shown the existence of several bugs. Proofs of the results presented in the paper have been omitted for the sake of brevity and can be found in the full version which is available upon request.

2 System Model

In this paper, we consider multi-threaded programs wherein threads communicate using locks. We model each thread using the trace flow graph framework (cf. [4]). Here each procedure of a thread is modeled as a flow graph, each node of which represents a control point of the procedure. The edges of the flow graph are annotated with statements that could either be assignments, calls to other procedures of the same thread or the acquire and release of locks when the thread needs to access shared resources. Recursion and mutual procedure calls are allowed. So that the flow graph of a program has a finite

number of nodes, abstract interpretation techniques are often used in order to get a finite representation of the (potentially infinitely many) control states of the original thread. This typically introduces non-determinism which is explicitly allowed. Each thread can then be modeled as a system of flow graphs representing its procedures.

The resulting framework of finite state flow graphs with recursion can be naturally modeled as a *pushdown system (PDS)*. A PDS has a finite control part corresponding to the valuation of the local variables of the procedure it represents and a stack which provides a means to model recursion.

Formally, a PDS is a five-tuple $\mathcal{P} = (P, Act, \Gamma, c_0, \Delta)$, where P is a finite set of *control locations*, *Act* is a finite set of *actions*, Γ is a finite *stack alphabet*, and $\Delta \subseteq (P \times \Gamma) \times Act \times (P \times \Gamma^*)$ is a finite set of *transition rules*. If $((p, \gamma), a, (p', w)) \in \Delta$ then we write $\langle p, \gamma \rangle \overset{a}{\hookrightarrow} \langle p', w \rangle$. A *configuration* of $\mathcal{P}$ is a pair $\langle p, w \rangle$, where $p \in P$ denotes the control location and $w \in \Gamma^*$ the *stack content*. We call c_0 the *initial configuration* of $\mathcal{P}$. The set of all configurations of $\mathcal{P}$ is denoted by $\mathcal{C}$. For each action a, we define a relation $\overset{a}{\rightarrow} \subseteq \mathcal{C} \times \mathcal{C}$ as follows: if $\langle q, \gamma \rangle \overset{a}{\hookrightarrow} \langle q', w \rangle$, then $\langle q, \gamma v \rangle \overset{a}{\rightarrow} \langle q', wv \rangle$ for every $v \in \Gamma^*$.

We model multi-threaded programs using PDSs communicating using locks. For a concurrent program comprised of threads T_1,...,T_n, a lock l is a globally shared variable taking on values from the set $\{1, ..., n, \perp\}$. The value of l can be modified by a thread using the operations *acquire(*l*)* and *release(*l*)*. A thread can acquire a lock l only if its value is currently $\perp$, viz., none of the other threads currently has possession of it. Once l has been acquired by thread T_i, its value is set to i and it remains so until T_i releases it by executing *release(*l*)* thereby resetting its value to $\perp$. Locks are not pre-emptible, viz., a thread cannot be forced to give up any lock acquired by it.

Formally, we model a concurrent program with n threads and m locks $\mathsf{l}_1, ..., \mathsf{l}_m$ as a tuple of the form $\mathcal{CP} = (T_1, ..., T_n, L_1, ..., L_m)$, where T_1,...,T_n are pushdown systems (representing threads) with the same set *Act* of non-*acquire* and non-*release* actions, and for each i, $L_i \subseteq \{\perp, 1, ..., n\}$ is the possible set of values that lock l_i can be assigned to. A global configuration of $\mathcal{CP}$ is a tuple $c = (t_1, ..., t_n, l_1, ..., l_m)$ where $t_1, ..., t_n$ are, respectively, the configurations of threads $T_1, ..., T_n$ and $l_1, ..., l_m$ the values of the locks. If no thread holds the lock in configuration c, then $l_i = \perp$, else l_i is the index of the thread currently holding the lock. The initial global configuration of $\mathcal{CP}$ is $(c_1, ..., c_n, \underbrace{\perp, ..., \perp}_{m})$, where c_i is the initial configuration of thread T_i. Thus all locks are *free* to start with. We extend the relation $\overset{a}{\longrightarrow}$ to pairs of global configurations as follows: Let $c = (c_1, ..., c_n, l_1, ..., l_m)$ and $c' = (c'_1, ..., c'_n, l'_1, ..., l'_m)$ be global configurations. Then

- $c \overset{a}{\longrightarrow} c'$ if there exists $1 \leq i \leq n$ such that $c_i \overset{a}{\longrightarrow} c'_i$, and for all $1 \leq j \leq n$ such that $i \neq j$, $c_j = c'_j$, and for all $1 \leq k \leq m$, $l_k = l'_k$.
- $c \overset{acquire(\mathsf{l}_i)}{\longrightarrow} c'$ if there exists $1 \leq j \leq n$ such that $c_j \overset{acquire(\mathsf{l}_i)}{\longrightarrow} c'_j$, and $l_i = \perp$, and $l'_i = j$, and for all $1 \leq k \leq n$ such that $k \neq j$, $c_k = c'_k$, and for all $1 \leq p \leq m$ such that $p \neq i$, $l_p = l'_p$.

– $c \xrightarrow{release(\mathsf{l}_i)} c'$ if there exists $1 \leq j \leq n$ such that $c_j \xrightarrow{release(\mathsf{l}_i)} c'_j$, and $l_i = j$, and $l'_i = \perp$, and for all $1 \leq k \leq n$ such that $k \neq j$, $c_k = c'_k$, and for all $1 \leq p \leq m$ such that $p \neq i$, $l_p = l'_p$.

A sequence $x = x_1, x_2, ...$ of global configurations of $\mathcal{CP}$ is a *computation* if x_1 is the initial global configuration of $\mathcal{CP}$ and for each i, $x_i \xrightarrow{a} x_{i+1}$, where either $a \in Act$ or for some $1 \leq j \leq m$, $a = release(\mathsf{l}_j)$ or $a = acquire(\mathsf{l}_j)$. Given a thread T_i and a reachable global configuration $c = (c_1, ..., c_n, l_1, ..., l_m)$ of $\mathcal{CP}$ we use *Lock-Set*(T_i, c) to denote the set of indices of locks held by T_i in c, viz., the set $\{j \mid l_j = i\}$.

Nested versus Non-nested Lock Access. We say that a concurrent program accesses locks in a nested fashion if and only if along each computation of the program a thread can only release the last lock that it acquired along that computation and that has not yet been released. For example in the figure below, the thread comprised of procedures `foo_nested` and `bar` accesses locks `a`,`b`, and `c` in a nested fashion whereas the thread comprised of procedures `foo_not_nested` and `bar` does not. This is because calling `bar` from `foo_non_nested` releases lock `b` before lock `a` even though lock `a` was the last one to be acquired.

Global Locks: `a,b,c`

```
foo_nested() {
   acquire(a);
   acquire(b);
   bar();
   release(c);
}
```

```
bar(){
   release(b);
   release(a);
   acquire(c);
}
```

```
foo_non_nested(){
   acquire(b);
   acquire(a);
   bar();
   release(c);
}
```

3 Many to Few

Let $\mathcal{CP}$ be a concurrent program comprised of n threads $T_1, ..., T_n$ and let f be a correctness property either of the form $\mathsf{E}_{\mathsf{fin}}h(i,j)$ or of the form $\mathsf{A}_{\mathsf{fin}}h(i,j)$, where $h(i,j)$ is an LTL\X formula with atomic propositions over the control states of threads T_i and T_j and $\mathsf{E}_{\mathsf{fin}}$ and $\mathsf{A}_{\mathsf{fin}}$ quantify solely over finite computation paths. Note that since $\mathsf{A}_{\mathsf{fin}}$ and $\mathsf{E}_{\mathsf{fin}}$ are dual path quantifiers it suffices to only consider the case where f is of the form $\mathsf{E}_{\mathsf{fin}}h(i,j)$. We show that in order to model check $\mathcal{CP}$ for f it suffices to model check the program $\mathcal{CP}(i,j)$ comprised solely of the threads T_i and T_j. We emphasize that this result does not require the given concurrent program to have nested locks. Formally, we show the following.

Proposition 1 (Double-Indexed Reduction Result). *Given a concurrent program $\mathcal{CP}$ comprised of n threads $T_1, ..., T_n$, $\mathcal{CP} \models \mathsf{E}_{\mathsf{fin}}h(i,j)$ iff $\mathcal{CP}(i,j) \models \mathsf{E}_{\mathsf{fin}}h(i,j)$, where $\mathcal{CP}(i,j)$ is the concurrent program comprised solely of the threads T_i and T_j.*

Proposition 2 (Single-Indexed Reduction Result). *Given a concurrent program $\mathcal{CP}$ comprised of n threads $T_1, ..., T_n$, $\mathcal{CP} \models \mathsf{E}_{\mathsf{fin}}h(i)$ iff $\mathcal{CP}(i) \models \mathsf{E}_{\mathsf{fin}}h(i)$, where $\mathcal{CP}(i)$ is the program comprised solely of the thread T_i.*

Similar results holds for properties of the form $\mathsf{E}_{\mathsf{inf}}h(i,j)$, where $\mathsf{E}_{\mathsf{inf}}$ quantifies solely over infinite computations.

4 Liveness Properties

Using proposition 2, we can reduce the model checking problem for a single-indexed LTL\X formula f for a system with n threads to a system comprised solely of the single thread whose control states are being tracked by f. Thus the problem now reduces to model checking a pushdown system for LTL\X properties which is known to be decidable in polynomial time in size of the control part of the pushdown system [2]. We thus have the following.

Theorem 3 *The model checking problem for single-indexed LTL\X properties for a system with n threads is decidable in polynomial time in the size of the PDS representing the thread being tracked by the property.*

5 Safety Properties

Even though proposition 2 allows us to reduce reasoning about double-indexed LTL\X properties from a system with n threads to one with 2 threads, it still does not yield decidability. This is because although the model checking of LTL\X is decidable for a single pushdown system, it becomes undecidable even for simple reachability and even for systems with only two PDSs communicating via pairwise rendezvous [11]. The proof of undecidability rests on the fact that synchronization using pairwise rendezvous couples the two PDSs tightly enough to allow construction of a system that accepts the intersection of the two given context free languages (CFLs) the non-emptiness of which is undecidable.

We show that if we allow PDSs with non-nested lock access then the coupling, though seemingly weaker than pairwise rendezvous, is still strong enough to build a system accepting the intersection of the CFLs corresponding to the given PDSs thus yielding undecidability for even pairwise reachability. This is discouraging from a practical standpoint. However we exploit the observation that in most real-world concurrent programs locks are accessed by threads in a nested fashion. In fact, in certain programming languages like Java and C#, locks are syntactically guaranteed to be nested. In that case, we can reduce the model checking of LTL\X properties in a sound and complete fashion for a concurrent program comprised of two threads to individually model checking augmented versions of the thread for LTL\X properties, which by [2] is efficiently decidable. Then combining this with the reduction result of the previous section, we get that the model checking problem of doubly-indexed LTL\X formulas is efficiently decidable for concurrent programs with nested locks.

5.1 Decidability of Pairwise Reachability for Nested Lock Programs

We motivate our technique with the help of a simple concurrent program $\mathcal{CP}$ shown below comprised of thread one with procedures `thread_one` and `acq_rel_c`, and thread two with procedures `thread_two` and `acq_rel_b`.

Suppose that we are interested in deciding whether $\mathsf{EF}(c4 \wedge g4)$ holds. The key idea is to reduce this to checking $\mathsf{EF}c4$ and $\mathsf{EF}g4$ individually on (modifications of) the two threads. Then given computations x and y leading to $c4$ and $g4$, respectively, we merge

them to construct a computation z of $\mathcal{CP}$ leading to a global configuration with threads one and two in local control states $c4$ and $g4$, respectively. Consider, for example, the computations x: c1,c2,d1,d2,c3,c4 and y: g1,g2,g3,h1,h2,g4 of threads one and two, respectively. Note that at control location g4, thread 2 holds locks c and d. Also, along computation y once thread two acquires lock c at control location g1, it does not release it and so we have to make sure that before we let it execute g1 along z, all operations that acquire and release lock c along x should already have been executed. Thus in our case g1 must be scheduled to fire only after d2 (and hence c1, c2 and d1) have already been executed. Similarly, operation c3 of thread one must be executed after h2 has already been fired along z. Thus one possible computation z of $\mathcal{CP}$ with the desired properties is z: c1, c2, d1, d2, g1, g2, g3, h1, h2, g4, c3, c4.

```
thread_one(){          acq_rel_c(){          thread_two(){
  c1: acquire(a) ;       d1: acquire(c) ;      g1: acquire(c) ;
  c2: acq_rel_c() ;      d2: release(c) ;      g2: acquire(d) ;
  c3: acquire(b) ;     }                       g3: acq_rel_b() ;
  c4: release(b) ;     acq_rel_b(){            g4: release(d) ;
  c5: release(a) ;       h1: acquire(b) ; }
}                        h2 release(b) ;
                       }
```

Note that if we replace the function call at control location g3 of thread two by acq_rel_a() which first acquires and then releases lock a, then there is no way to reconcile the two local computations x and y to get a global computation leading to a configuration with threads one and two, respectively, at control locations c4 and g4, even though they are reachable in their respective individual threads. This is because in this case h2 (and hence g1, g2, g3 and h1) should be executed before c1 (and hence c2, d1, d2, c3 and c4). Again, as before, g1 can be fired only after d2 (and hence c1, c2 and d1). From the above observations we get that g1 must be fired after h2 along z. But that violates the local ordering of the transitions fired along y wherein g1 was fired before h2. This proves the claim made above.

In general when testing for reachability of control states c and c' of two different threads it suffices to test whether there exist paths x and y in the individual threads leading to states c and c' holding lock sets L and L' which can be acquired in a compatible fashion. Compatibility ensures that we do not get a scenario as above where there exist locks $a \in L$ and $a' \in L'$ such that a transition acquiring a' was fired after acquiring a for the last time along x and a transition acquiring a was fired after acquiring a' for the last time along y, else we can't reconcile x and y. The above discussion is formalized below in Theorem 5. Before proceeding further, we need the following definition.

Definition 4 (Acquisition History). *Let x be a global computation of a concurrent program $\mathcal{CP}$ leading to global configuration c. Then for thread T_i and lock l_j of $\mathcal{CP}$ such that $j \in$ Lock-Set(T_i, c), we define $AH(T_i, l_j, x)$ to be the set of indices of locks that were acquired (and possibly released) by T_i after the last acquisition of l_j by T_i along x.*

Theorem 5 (Decomposition Result). *Let $\mathcal{CP}$ be a concurrent program comprised of the two threads T_1 and T_2 with nested locks. Then for control states a_1 and b_2 of T_1 and T_2, respectively, $\mathcal{CP} \models \mathsf{EF}(a_1 \wedge b_2)$ iff there are computations x and y of the individual threads T_1 and T_2, respectively, leading to configurations s with T_1 in control state a_1 and t with T_2 in control state b_2 such that*

- Lock-Set$(T_1, s) \cap$ Lock-Set$(T_2, t) = \emptyset$.
- *there do not exist locks $l \in$ Lock-Set(T_1, s) and $l' \in$ Lock-Set(T_2, t) with $l' \in AH(T_1, l, x)$ and $l \in AH(T_2, l', y)$.*

To make use of the above result we augment the given threads to keep track of the acquisition histories. Given a thread $\mathcal{P} = (P, Act, \Gamma, c_0, \Delta)$ of concurrent program $\mathcal{CP}$ having the set of locks L of cardinality m, we construct the augmented thread $\mathcal{P}_A = (P_A, Act, \Gamma, d_0, \Delta_A)$, where $P_A = P \times 2^L \times (2^L)^m$ and $\Delta_A \subseteq (P_A \times \Gamma) \times (P_A \times \Gamma^*)$. The augmented PDA is used to track the set of locks and acquisition histories of thread T along local computations of T. Let x be a computation of $\mathcal{CP}$ leading to global configuration s. Each control location of the augmented PDA is of the form $(a, Locks, AH_1, ..., AH_m)$, where a denotes the current control state of T in s, $Locks$ the set of locks currently held by T and for $1 \leq j \leq m$, if $l_j \in Locks$, then AH_j is the set $AH(T, l_j, x)$ else it is the empty set. The initial configuration d_0 is the $(m+2)$-tuple $(c_0, \emptyset, \emptyset, ..., \emptyset)$. The transition relation Δ_A is defined as follows:

$\langle q, Locks, AH_1, ..., AH_m, \gamma\rangle \overset{a}{\hookrightarrow} \langle q', Locks', AH'_1, ..., AH'_m, w\rangle \in \Delta_A$ iff

- a is not a lock operation, $\langle q, \gamma\rangle \overset{a}{\hookrightarrow} \langle q', w\rangle \in \Delta$, $Locks = Locks'$ and for $1 \leq j \leq m$, $AH_j = AH'_j$.
- a is the action $acquire(l_k)$, $Locks' = Locks \cup \{k\}$, $q \overset{acquire(l_k)}{\longrightarrow} q'$, $\gamma = w$ and for $1 \leq p \leq m$, if $p \in Locks$ then $AH'_p = AH_p \cup \{k\}$ and $AH'_p = AH_p$ otherwise.
- a is the action $release(l_k)$, $Lock' = Locks \setminus \{k\}$, $q \overset{release(l_k)}{\longrightarrow} q'$, $\gamma = w$, $AH'_k = \emptyset$ and for $1 \leq p \leq m$ such that $p \neq k$, $AH'_p = AH_p$.

Proposition 6. *Given a concurrent program $\mathcal{CP}$ comprised of threads T and T', the model checking problem for pairwise reachability, viz., formulas of the form $\mathsf{EF}(a_1 \wedge b_2)$ is decidable.*

Implementation Issues. Given a concurrent program, to implement our technique we introduce for each lock `l` two extra global variables defined as follows:

1. `possession_l`: to track whether `l` is currently in the possession of a thread
2. `history_l`: to track the acquisition history of `l`.

To begin with, `possession_l` is initialized to *false* and `history_l` to the emptyset. Then each statement of the form `acquire(lk)` in the original code is replaced by the following statements:

```
acquire(lk) ;
possession_lk := true ;
for each lock l
    if (possession_l = true)
       history_l := history_l ∪ {lk} ;
```

Similarly, each statement of the form `release(lk)` is replaced with the following sequence of statements:

```
release(lk) ;
possession_lk := false ;
history_lk := emptyset ;
```

Optimizations. The above naive implementation keeps the acquisition history for each lock of the concurrent program and tests for all possible disjoint pairs L and L' of lock sets and all possible compatible acquisition histories at two given error control locations a_i and b_j, say. In the worst case this is exponential in the number of locks. However by exploiting program analysis techniques one can severely cut down on the number of such lock sets and acquisition histories that need to be tested for each control location of the given program as discussed below.

Combining Lock Analysis with Program Analysis. Using static analysis on the control flow graph of a given thread we can get a conservative estimate of the set of locks that could possibly have been acquired by a thread with its program counter at a given control location c. This gives us a superset L_c of the set of locks that could possibly have been acquired at control location c and also the possible acquisition histories. Thus in performing the reachability analysis, $\mathsf{EF}(a_i \wedge b_j)$, we only need to consider sets L and L' of locks such that $L \cap L' = \emptyset$, $L \subseteq L_{a_i}$ and $L' \subseteq L_{b_j}$. This can exponentially cut down on the lock sets and acquisition histories that need be explored as, in practice, the nesting depth of locks is usually one and so the cardinality of L_c will usually be one.

Combining Lock Analysis with Program Slicing. By theorem 5, for a control location c of thread T we need to track histories of only those locks that are in possession of T at c instead of every lock as was done in the naive implementation. Furthermore for a lock l in possession of T at c we can ignore all lock operations performed by T before l was acquired for the last time by T before reaching c as these don't affect the acquisition history of l. Such statements can thus be deleted using program slicing techniques.

6 Deadlockability

In our framework, since synchronization among threads is carried out using locks, the only way a deadlock can occur is if there is a reachable global state s with a dependency cycle of the form $T_{i_1} \rightarrow T_{i_2} \rightarrow ... \rightarrow T_{i_p} \rightarrow T_{i_1}$, where $T_{i_{k-1}} \rightarrow T_{i_k}$ if the program counter of $T_{i_{k-1}}$ is currently at an acquire operation for a lock that is currently held by T_{i_k}. Thus to decide whether any thread in the given program $\mathcal{CP}$ is deadlockable, for each thread T_i of $\mathcal{CP}$ we first construct the set of reachable configurations of the corresponding augmented thread $(T_i)_A$ (defined above) where the control location is such that an acquire operation can be executed from it. Denote the set of such configurations by Acq_i. We then construct a directed graph $D_{\mathcal{CP}}$ whose nodes are elements of the set $\bigcup_i Acq_i$ and there is a directed edge from configuration $\mathbf{a} = (a, L, AH_1, ..., AH_m)$ to $\mathbf{a'} = (a', L', AH'_1, ..., AH'_m)$ iff there exist $i \neq i'$ such

that (i) $\mathbf{a} \in Acq_i$, $\mathbf{a}' \in Acq_{i'}$, and (ii) a and a' both correspond to acquire operations, say, $acquire(l)$ and $acquire(l')$, respectively, and (iii) thread $T_{i'}$ currently holds lock l required by thread T_i, viz., $l \in L'$. Then the given current program is deadlockable iff there exists a cycle $\mathbf{c}_1 \rightarrow ... \rightarrow \mathbf{c}_p \rightarrow \mathbf{c}_1$ in $D_{\mathcal{CP}}$ such that every pair of configurations $\mathbf{c}_j = (c_j, L_j, AH_{j1}, ..., AH_{jm})$ and $\mathbf{c}_{j'} = (c_{j'}, L_{j'}, AH_{j'1}, ..., AH_{j'm})$ is consistent, viz., $L_j \cap L_{j'} = \emptyset$ and there do not exist locks $l_r \in L_j$ and $l_{r'} \in L_{j'}$ such that $l_{r'} \in AH_{jr}$ and $l_r \in AH_{j'r'}$. By theorem 5, consistency ensures that the global state encompassing the cycle is a reachable state of $\mathcal{CP}$. Note that again we have bypassed the state explosion problem by not constructing the state space of the system at hand. Furthermore using the optimizations discussed in the previous section, we can severely cut down on the possible lock sets and acquisition histories that we need to track for each acquire location in each thread. This ensures that the size of $D_{\mathcal{CP}}$ remains tractable.

7 Undecidability for Programs with Non-nested Locks

In this section, we show that for concurrent programs comprised of two threads T_1 and T_2 communicating via locks (not necessarily nested), the model checking problem for *pairwise reachability*, viz., properties of the form $\mathsf{EF}(a_1 \wedge b_2)$, where a_1 and b_2 are control states of T_1 and T_2, respectively, is undecidable.

Given a concurrent program $\mathcal{CP}$ comprised of two threads T_1 and T_2 communicating via pairwise rendezvous, we construct a new concurrent program $\mathcal{CP}'$ comprised of threads T_1 and T_2 by (weakly) simulating pairwise rendezvous using non-nested locks such that the set of control states of T_1 and T_2 are supersets of the sets of control states of T_1 and T_2, respectively, and for control states a_1 and b_2 of T_1 and T_2, respectively, $\mathcal{CP} \models \mathsf{EF}(a_1 \wedge b_2)$ iff $\mathcal{CP}' \models \mathsf{EF}(a_1 \wedge b_2)$. This reduces the decision problem for pairwise reachability for threads communicating via pairwise rendezvous to threads communicating via locks. But since pairwise reachability for threads communicating via pairwise rendezvous is undecidable, our result follows.

Simulating Pairwise Rendezvous using Locks. We now present the key idea behind the simulation. We show how to simulate a given pair $a \xrightarrow{m!} b$ and $c \xrightarrow{m?} d$ of send and receive pairwise rendezvous transitions, respectively. Recall that for this rendezvous to be executed, both the send and receive transitions must be simultaneously enabled, else neither transition can fire. Corresponding to the labels $m!$ and $m?$, we first introduce the new locks $l_{m!}$, $l_{m?}$ and l_m.

Cons ider now the send transition $tr : a \xrightarrow{m!} b$ of thread T_1, say. Our construction ensures that before T_1 starts mimicking tr in local state a it already has possession of lock $l_{m?}$. Then T_1 simulates T_1 via the following sequence of transitions: $a \xrightarrow{acquire(l_m)} a_1 \xrightarrow{release(l_{m?})} a_2 \xrightarrow{acquire(l_{m!})} a_3 \xrightarrow{release(l_m)} b \xrightarrow{acquire(l_{m?})} b_1 \xrightarrow{release(l_{m!})} b_2$. Similarly we assume that T_2 has possession of $l_{m!}$ before it starts mimicking $tr' : c \xrightarrow{m?} d$. Then T_2 simulates tr' by firing the following sequence of transitions: $c \xrightarrow{acquire(l_{m?})} c_1 \xrightarrow{release(l_{m!})} c_2 \xrightarrow{acquire(l_m)} c_3 \xrightarrow{release(l_{m?})} d \xrightarrow{acquire(l_{m!})} d_1 \xrightarrow{release(l_m)} d_2$.

The reason for letting thread T_1 acquire $l_{m?}$ at the outset is to prevent thread T_2 from firing transition $c \xrightarrow{m?} d$ without synchronizing with $tr : a \xrightarrow{m!} d$. To initiate execution of the pairwise rendezvous involving tr, thread T_1 releases lock $l_{m?}$ and only when lock $l_{m?}$ is released can T_2 pick it up in order to execute the matching receive transition labeled with $m?$. But before T_1 releases $l_{m?}$ it acquires l_m. Note that this trick involving *chaining* wherein before releasing a lock a thread is forced to pick up another lock gives us the ability to introduce a relative ordering on the firing of local transitions of T_1 and T_2 which in turn allows us to simulate (in a weak sense) the firing of the pairwise rendezvous comprised of tr and tr'. It can be seen that due to chaining, the local transitions in the two sequences defined above can only be interleaved in the following order: $a \xrightarrow{acquire(l_m)} a_1, a_1 \xrightarrow{release(l_{m?})} a_2, c \xrightarrow{acquire(l_{m?})} c_1, c_1 \xrightarrow{release(l_{m!})} c_2$, $a_2 \xrightarrow{acquire(l_{m!})} a_3\ a_3 \xrightarrow{release(l_m)} b, c_2 \xrightarrow{acquire(l_m)} c_3, c_3 \xrightarrow{release(l_{m?})} d, b \xrightarrow{acquire(l_{m?})} b_1$, $b_1 \xrightarrow{release(l_{m!})} b_2, d \xrightarrow{acquire(l_{m!})} d_1, d_1 \xrightarrow{release(l_m)} d_2$.

It is important to note that the use of overlapping locks is essential in implementing chaining thereby forcing a pre-determined order of firing of the local transitions which cannot be accomplished using nested locks alone. Since the model checking problem for pairwise reachability is known to be undecidable for threads communicating using pairwise rendezvous [11] and since we can, by the above result, simulate pairwise rendezvous using locks in a way so as to preserve pairwise reachability, we have the following undecidability result.

Theorem 8. *The model checking problem for pairwise reachability is undecidable for concurrent programs comprised of two threads communicating using locks.*

8 The Daisy Case Study

We have used our technique to find bugs in the Daisy file system which is a benchmark for analyzing the efficacy of different methodologies for verifying concurrent programs [1]. Daisy is a 1KLOC Java implementation of a toy file system where each file is allocated a unique inode that stores the file parameters and a unique block which stores data. An interesting feature of Daisy is that it has fine grained locking in that access to each file, inode or block is guarded by a dedicated lock. Moreover, the acquire and release of each of these locks is guarded by a 'token' lock. Thus control locations in the program might possibly have multiple open locks and furthermore the acquire and release of a given lock can occur in different procedures.

We have incorporated our lock analysis technique into F-Soft [8] which is a framework for model checking sequential software. We have implemented the decision procedure for pairwise reachability and used it to detect race conditions in the Daisy file system. Towards that end we check that for all n, any n-threaded Daisy program does not have a given race condition. Since a race condition can be expressed as pairwise reachability, using Proposition 1, we see that it suffices to check a 2-thread instance. Currently F-Soft only accepts programs written in C and so we first manually translated the Daisy code which is written in Java into C. Furthermore, to reduce the model sizes, we truncated the sizes of the data structures modeling the disk, inodes, blocks,

file names, etc., which were not relevant to the race conditions we checked, resulting in a sound and complete *small-domain* reduction. We emphasize that beyond redefining the constants limiting these sizes no code restructuring was carried out on the translated C code.

Given a race condition to be verified, we use a fully automated procedure to generate two augmented thread representation (Control Flow Graphs (CFG)) on which the verification is carried out individually. A race condition occurs if and only if the labels in both the modified threads are reachable. Using this fully automated procedure, we have shown the existence of the following race conditions also noted by other researchers (cf. [1]):

1. Daisy maintains an allocation area where for each block in the file system a bit is assigned 0 or 1 accordingly as the block has been allocated to a file or not. But each disk operation reads/writes an entire byte. Two threads accessing two different files might access two different blocks. However since bytes are not guarded by locks in order to set their allocation bits these two different threads may access the same byte in the allocation block containing the allocation bit for each of these locks thus setting up a race condition. For the data race described above, the statistics are as follows. The pre-processing phase which includes slicing, range analysis, using static analysis to find the possible set of open locks at the control state corresponding to the error label and then incorporating the acquisition history statements in the CFGs corresponding to the threads for only these locks took 77 secs[2] for both the threads. The two model checking runs took 5.3 and 21.67 secs and the error labels corresponding to the race condition were reached at depths 75 and 333, respectively in the two threads using SAT-based BMC in F-Soft.

2. In Daisy reading/writing a particular byte on the disk is broken down into two operations: a seek operation that mimics the positioning of the head and a read/write operation that transfers the actual data. Due to this separation between seeking and data transfer a race condition may occur. For example, reading two disk locations, say n and m, we must make sure that $seek(n)$ is followed by $read(n)$ without $seek(m)$ or $read(m)$ scheduled in between. Here the pre-processing phase took about the same time as above. The model checking runs on the two threads took 15 and 35 secs.

9 Conclusion and Related Work

In this paper, we have considered the static analysis of concurrent multi-threaded programs wherein the threads communicate via locks. We have shown that for single index LTL\X properties the problem is efficiently decidable. On the other hand, for double index LTL\X properties, the problem can be shown to be undecidable even for reachability and for systems with only two threads. However, for the practically important case where the locks are nested we get efficient decidability for pairwise reachability. We have implemented our technique in a prototype software verification platform and our preliminary results on the Daisy benchmark are encouraging.

There has been interesting prior work on extending data flow analysis to handle concurrent programs. In [3], the authors attempt to generalize the decision procedures given

[2] Machine Specifications: Intel Pentium4 3.20GHz CPU, 2MB RAM.

in [2] to handle pushdown systems communicating via CCS-style pairwise rendezvous. However since even reachability is undecidable for such a framework the procedures are not guaranteed to terminate in general but only for certain special cases, some of which the authors identify. However their practical utility is not clear. The key idea in identifying the above cases was to restrict the interaction among the threads so as to bypass the undecidability barrier. A natural way to accomplish that which was formulated in [10] is to explore the state spaces of the concurrent program for a bounded number of context switches among the threads.

A commonly used approach to cut down on the number of interleavings when reasoning about concurrent systems is to exploit syntactic independence among the local operations of different components of the system. In [9] this idea is exploited using the concept of transactions, wherein executing a transaction is equivalent to atomically executing a sequence of operations of each thread that do not involve communication with other threads and thus their execution does not interfere in the operation of other threads. The advantage of this technique is that it works well provided one can statically decide whether two given operations from two different threads are independent, a problem which is, in general, hard. The disadvantage is that although this technique can potentially cut down on the number of interleavings to be explored, it still does not completely address the core issue of state explosion. The use of partial order techniques ([6, 5, 14, 15]) also exploits syntactic independence of transitions to cut down on the number of interleavings to be explored and although extremely useful, suffers from the same drawback as above.

Another technique that has been adapted from concurrent protocol verification is the use of compositional reasoning wherein one tries to reduce reasoning about the correctness of a system comprised of many concurrently executing components to reasoning about each individual components. In [7] an assume-guarantee style reasoning is used to abstract out the environment of each thread in a purely automatic fashion for system where the threads are loosely coupled. A drawback is that the technique is not complete for reachability and thus is not guaranteed to find all errors. Furthermore, error trace recovery is hard because abstracting the environment causes a lot of information to be lost and thus it may not be possible to construct a concrete error trace purely from the environment assumptions.

We, on the other hand, have identified a practically important case of threads communicating using locks and shown how to reason efficiently about a rich class of properties. We address the state explosion problem by reducing reasoning about indexed LTL\X properties and deadlockability to reasoning about individual threads. Our methods are exact, i.e., both sound and complete, and cater to automatic error trace recovery. A key advantage of our method is that by avoiding construction of the state space of the system at hand we bypass the state explosion problem, thus guaranteeing scalability of our approach. Most multi-threaded programs use shared data structures that are guarded by locks to communicate. A potential drawback of our method is that it works only for threads that communicate purely using locks. However we believe that a very large fraction of concurrent software is loosely coupled and even where richer communication mechanisms are used, the interaction between the threads is not very subtle and can often, by using standard abstract interpretation techniques, be modeled

as threads communicating solely using locks. Furthermore, even if it not possible to abstract out the shared data structures, by considering only communication via the locks we over-approximate the set of behaviors of the given program. Our technique can then be used to generate warnings for potential data race violations. This is advantageous as model checking a single thread is more tractable than model checking an entire multi-threaded program. Once these potential data race violations have been isolated, more general but less tractable techniques like model checking using partial order reductions can be deployed to further refine the analysis by focusing precisely on these violations. Our technique can therefore be looked upon as a first line of attack in combating state explosion in the context of multi-threaded software providing an exact and efficient procedure for verifying an important class of multi-threaded software.

References

1. Joint CAV/ISSTA Special Event on Specification, Verification, and Testing of Concurrent Software. In *http://research.microsoft.com/ qadeer/cav-issta.htm*.
2. Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability Analysis of Pushdown Automata: Application to Model-Checking. In *CONCUR*, LNCS 1243, pages 135–150, 1997.
3. Ahmed Bouajjani, Javier Esparza, and Tayssir Touili. A generic approach to the static analysis of concurrent programs with procedures. In *IJFCS*, volume 14(4), pages 551–, 2003.
4. M. B. Dwyer and L. A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *ACM SIGSOFT*, pages 62–75, 1994.
5. P. Godefroid. Model Checking for Programming Languages using Verisoft. In *POPL*, pages 174–186, 1997.
6. P. Godefroid and P. Wolper. Using Partial Orders for Efficient Verification of deadlock-freedom and safety properties. In *Formal Methods in Systems Design*, pages 149–164, 1993.
7. T. Henzinger, R. Jhala, R. Mazumdar, and S. Qadeer. Thread-Modular Abstraction Refinement. In *CAV*, LNCS 2725, pages 262–274, 2003.
8. F. Ivančić, Z. Yang, M. Ganai, A. Gupta, and P. Ashar. Efficient SAT-based Bounded Model Checking for Software Verification. In *Symposium on Leveraging Applications of Formal Methods*, 2004.
9. S. Qadeer, S. K. Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *POPL*, pages 245–255, 2004.
10. S. Qadeer and J. Rehof. Context-Bounded Model Checking of Concurrent Software. In *TACAS*, 2005.
11. G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. In *ACM Trans. Program. Lang. Syst.*, volume 22(2), pages 416–430, 2000.
12. T. W. Reps, S. Horwitz, and S. Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL*, pages 49–61, 1985.
13. D. A. Schmidt and B. Steffen. Program Analysis as Model Checking of Abstract Interpretations. In *Static Analysis, 5th International Symposium,*, LNCS 1503, pages 351–380, 1998.
14. S. D. Stoller. Model-Checking Multi-Threaded Distributed Java Programs. In *STTT*, volume 4(1), pages 71–91, 2002.
15. W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model Checking Programs. In *Automated Software Engineering*, volume 10(2), pages 203–232, 2003.
16. I. Walukeiwicz. Model Checking CTL Properties of Pushdown Systems. In *FSTTCS*, LNCS 1974, pages 127–138, 2000.