Automatic Lock Insertion in Concurrent Programs

Vineet Kahlon, NEC Labs, Princeton, USA.

Abstract—Triggering errors in concurrent programs is a notoriously difficult task. A key reason for this is the behavioral complexity resulting from the large number of interleavings of operations of different threads. An even more challenging task is fixing errors once they are detected. In general, automatically synthesizing a correct program from a buggy one is a hard problem. However for simple correctness properties that depend on the syntactic structure of the program rather than its semantics, automatic error correction becomes feasible. In this paper, we consider the problem of lock insertion to enforce critical sections required to fix bugs like atomicity violations. A key challenge in lock insertion is that enforcing critical sections is not the sole criterion that needs to be satisfied. Often other correctness constraints like deadlock-freedom also need to be met. Moreover, apart from ensuring correctness, another key concern during lock insertion is performance. Indeed, mutual exclusion constraints generated by locks kill parallelism thereby impacting performance. Thus it is crucial that the newly introduced critical sections be kept as small as possible. In other words, our goal is lock insertion while meeting the dual, and often conflicting, requirements of (i) correctness and (ii) performance. In this paper, we present a fully automatic, provable optimal, efficient and precise technique for lock insertion in concurrent code that ensures deadlock freedom while attempting to minimize the resulting critical sections.

I. INTRODUCTION

Detecting errors in concurrent programs is a notoriously difficult task. A key reason for this is the behavioral complexity resulting from the large number of interleavings of different threads. An even more challenging task is fixing errors once they are detected. In general, automatically synthesizing a correct program from a buggy one is hard. However for simple correctness properties that depend on the syntactic structure of the program rather than its semantics, automatic error correction becomes feasible. An example is the insertion of mutexes in order to enforce critical sections to fix data races or atomicity violations. Inserting mutexes typically does not require reasoning about program semantics but relies merely on aliasing information in order to identify sections of code with shared variable accesses that need to be executed atomically.

In this paper, we consider the problem of lock insertion to enforce critical sections required to fix bugs like atomicity violations. This can be accomplished in a trivial manner by simply encapsulating the desired regions of code within lock/unlock statements. However, enforcing critical sections is often not the sole criterion to be satisfied during lock insertion. Indeed, adding mutexes may introduce new deadlocks. Thus a key goal is to guarantee deadlock-free lock insertion, i.e., no *new* deadlocks are introduced.

Apart from ensuring correctness, another key concern during lock insertion is performance. Mutual exclusion constraints generated by locks kill parallelism thereby impacting performance. Thus it is critical that the newly introduced critical sections be kept as small as possible.

It is worth mentioning that there exist techniques in the literature [4], [12], [6], [1], [2] for lock insertion in programs without prior locks. However, this problem is easier than the one we consider in this paper, as for programs without locks deadlocks can be avoided simply by acquiring all locks in a pre-defined order. One way to handle lock insertion in programs with prior locks would be to first remove all pre-existing locks and then leverage existing lock insertion techniques. This approach, however, presents many practical obstacles.

First, before removing existing locks we would have to identify all pairs of mutually atomic segments of the form (s_1, s_2), where atomic segments s_1 and s_2 are guarded by the same lock. However, lock/unlock APIs typically take pointers to locks as parameters and so a whole program points-to analysis would be required in order to determine the locks guarding segments s_1 and s_2. Moreover, since lock pointers often point to different locks in different function calling contexts, this points-to analysis needs to be context-sensitive. However, it is well known that scaling a precise context-sensitive points-to analysis for large realistic programs comprised of multiple code modules is a non-trivial task.

Moreover, even after aliases have been computed precisely, it is not enough to enumerate all pairs of the form (s_1, s_2), where s_1 and s_2 are guarded with the same lock. This is because it is often the case that locks are re-used (to reduce their number in certain applications) so that segments s_1 and s_2 may be guarded with the same lock even though they may not execute in parallel. Thus in order to isolate all pairs of segments that are truly mutually atomic, we would need to (1) understand the reasons for introducing prior locking statements, i.e., be somewhat knowledgeable about the program's semantics which is not feasible for large applications, and (2) need at least a whole program MHP (may-happens-in-parallel) analysis to determine whether s_1 and s_2 can execute in parallel - expensive for large programs. Finally, we may end up having a large number of mutually atomic pairs of segments impacting scalability of lock insertion.

On the other hand, it is highly desirable that our lock insertion technique avoids a whole program analysis and restricts the analysis (including the context-sensitive points-to analysis) to only the few modules requiring code modification, i.e., where bugs have been detected. This is precisely what our lock insertion technique accomplishes. An important side benefit is that it ensures scalability of our analysis.

To sum up, our goal is a *localized* analysis for lock insertion meeting the dual constraints of (i) correctness and (ii) performance. These constraints are often conflicting in nature. Indeed, during lock insertion one of the key properties that we want to ensure is deadlock-freedom while keeping the critical sections as small as possible. If either one of these two requirements is dropped, then the problem is greatly simplified. For instance, if we give up the requirement of deadlock freedom then given a pair of code segments s_1 and s_2

to be executed in a mutually atomic fashion, it suffices to insert lock (unlock) statements for a new lock l, immediately before (after) the two segments in both threads. Clearly this induces minimal critical sections but does not guarantee that no new deadlocks have been introduced. Similarly, if the requirement of optimality is dropped then it suffices to introduce lock (unlock) statements for a new lock l at the last lock free states before (after) the segments in either thread. Such a solution ensures that no new deadlocks are introduced but may not be optimal.

Given a pair of mutually atomic code segments s_1 and s_2 in two different threads T_1 and T_2, respectively, of an n-thread program, we present a lock insertion strategy that involves a series of local moves that re-locates the newly inserted lock statements in the individual threads T_1 and T_2 in a dovetailed fashion till we achieve deadlock freedom. The interesting, and somewhat surprising, result is that our objective of minimizing the newly introduced critical sections which is inherently global in nature can be achieved via purely local moves of the locking statements in the individual threads. This is crucial as it allows our strategy to be *compositional* in nature, i.e., based only on thread local reasoning, thereby ensuring scalability.

While our lock insertion strategy is applicable to programs with arbitrary locking patterns, for implementation purposes we consider the special case of programs with *nested* locks. The main motivation for this is that almost all lock usage in real life programs is nested. Additionally, nested locks offer a key advantage in that they allow us to leverage the framework of acquisition histories [10] to formulate a provable efficient and compositional (thread local) analysis for lock insertion.

We demonstrate the efficacy of our technique on a broad range of benchmarks.

II. Program Model

We consider concurrent imperative programs comprised of threads that communicate using shared variables and synchronize with each other using standard primitives such as locks and rendezvous. Formally, we define a concurrent program $\mathcal{CP}$ as a tuple $(\mathcal{T}, \mathcal{V}, \mathcal{R}, s_0)$, where $\mathcal{T} = \{T_1, ..., T_n\}$ denotes a finite set of threads, $\mathcal{V} = \{v_1, ..., v_m\}$ a finite set of shared variables and synchronization objects with v_i taking on values from the set V_i, $\mathcal{R}$ the transition relation and s_0 the initial state of $\mathcal{CP}$. Each thread T_i is represented by the control flow graph of the sequential program it executes, and is denoted by the pair (C_i, R_i), where C_i denotes the set of control locations of T_i and R_i its transition relation. A global state s of $\mathcal{CP}$ is a tuple $(s[1], ..., s[n], v[1], ..., v[m]) \in \mathcal{S} = C_1 \times ... \times C_n \times V_1 \times ... \times V_m$, where $s[i]$ represents the current control location of thread T_i and $v[j]$ the current value of variable v_j. The global state transition diagram of $\mathcal{CP}$ is defined to be the standard interleaved parallel composition of the transition diagrams of the individual threads.

III. Lock Insertion Problem

The goal of lock insertion is to remove data races or, more generally, atomicity violations by enforcing critical sections that envelope regions of code to be executed atomically. These critical section may comprise multiple regions of contiguous code that we refer to as *atomic segments*. Due to branching

```
T1 () {
0a:  ...
1a:  while(sh    > 0) {
2a:    sh++;
3a:    ...
4a:  }
}
```

```
T2 () {
0b:  ...
1b:  sh = sh + 2;
2b:  ...
}
```

Fig. 1.   Split Critical Section

loops and recursion, a code segment of thread T is, in general, defined by a sub-graph of the CFG of T.

As an example, consider the threads T_1 and T_2 shown in Fig. 1 accessing shared variable sh. In thread T_1, due to the presence of a loop the critical section is broken up into two segments, one comprising the statements 1a and 2a and the other comprising the statement 4a. Note that we need to include 4a in the critical section because the condition of the while loop accesses the shared variable sh and we need to re-acquire the lock guarding access to 1a (in case it was released within the loop body) if we re-enter the loop body. The critical section in thread T_2, however, consists of only one atomic segment, i.e., 1b.

We define an *atomic segment* of thread T as the set of control locations occurring in a directed acyclic graph (DAG) whose (i) *roots*, i.e., nodes of in-degree zero, define the control locations of T marking the start of the segment, (ii) *leaves* define control locations marking the ends of the segment, and (iii) the successors of each location c in the segment are the non-backedge (as defined by some dfs ordering) successors of c in the CFG of T. We use $[(r_1, ..., r_p), (l_1, ..., l_q)]$ to denote an atomic segment with roots $r_1, ..., r_p$ and leaves $l_1, ..., l_q$. For example, $[1a, 2a]$ (or more precisely $[(1a), (2a)]$) denotes an atomic segment of T_1 in Fig. 1.

Whether a region of code in a thread is an atomic segment depends on the values of program counters of other threads. Indeed regions of code in different threads accessing the same shared variable need to be executed atomically *relative* to each other, while regions of code accessing different shared variables need not. This leads to the notion of mutually atomic segments.

Definition (Mutually Atomic Segments). *We say that code segments s_1 and s_2 of threads T_1 and T_2, respectively, are mutually atomic if there does not exist a reachable global state of the given concurrent program with T_1 and T_2 at control locations c_1 and c_2 occurring along segments s_1 and s_2, respectively.*

The lock insertion problem is then defined as follows.

Lock Insertion Problem. *Let $P = \{(s_1^1, s_1^2), ..., (s_k^1, s_k^2)\}$ be a set of pairs (s_j^1, s_j^2) of atomic segments s_j^1 and s_j^2. Identify locations in threads $T_1, ..., T_n$ comprising the given concurrent program to insert locks that guarantees the following*

1) *for each j, s_j^1 and s_j^2 are mutually atomic,*
2) *no new deadlocks are introduced, and*
3) *minimality of the newly introduced critical sections, as determined by the set of program statements in the critical sections.*

Conditions 1, 2 and 3 are collectively referred to as *Lock Insertion Requirements*. It is worth pointing out that our notion of minimality for critical sections is based on set inclusion as

opposed to the number of program statements comprising the critical section. This is the best one can hope for.

Consistency Invariant. In Fig. 1, we observe that the critical section in T_1, was 'split' into the segments $[1a, 2a]$ and $[4a, 4a]$ to maintain the consistency invariant that an un-acquired lock cannot be released (in case we re-enter the loop). We therefore assume that the atomic segments in the specification of the given lock insertion problem instance satisfy the following natural condition.

Consistency Invariant. *Let* $P = \{(s_1^1, s_1^2), ..., (s_k^1, s_k^2)\}$ *be a lock insertion problem instance, where for each j, s_j^1 and s_j^2 are the desired mutually atomic segments. Then if a loop head (tail) occurs in an atomic segment s_j^m comprising a critical section cs of thread T_i then its matching loop tail (head) also occurs in an (possibly the same) atomic segment $s_{j'}^{m'}$ comprising cs.*

IV. LOCK INSERTION

We start by observing that it suffices to formulate the lock insertion procedure for the case where we are given a single pair (CS_1, CS_2) of mutually atomic segments, where CS_1 and CS_2 are atomic segments in two different threads. The case where we are given multiple mutually atomic segment pairs can be handled by repeatedly applying the lock insertion procedure.

For ease of exposition, we start with the assumption that the threads are specified as straight line code with the general case being considered in sec V. The straight-line case suffices to show case the key ideas behind our lock insertion technique.

Let the given concurrent program be comprised of the threads $T_1, ..., T_k$ and let atomic segments CS_1 and CS_2 belong to threads T_1 and T_2. Suppose that threads T_1 and T_2 are defined via the sequences of control locations $T_1 : c_0, ..., c_n$ and $T_2 : d_0, ..., d_m$, respectively.

For the case where thread T is specified as the straight-line code $T : d_0, ..., d_p$, a segment s defining a critical section of T can be identified uniquely by its start and end locations d_i and d_j, respectively, where $i < j$. We denote such a segment by $s = [d_i, d_j]$, where $[d_i, d_j]$ denotes the set of control locations occurring between (and including) d_i and d_j along T.

Let the segments s_1 and s_2 of threads T_1 and T_2 be denoted by $s_1 = [c_i, c_j]$ and $s_2 = [d_{i'}, d_{j'}]$, respectively, where $i < j$ and $i' < j'$. Our goal is to introduce locking and unlocking statements *lock(l)* and *unlock(l)* for a *new* lock l, respectively, such that the lock insertion requirements are met.

Notation. Before proceeding further, we fix some notation. The locking statements *lock(l)* and *unlock(l)* inserted in threads T_1 and T_2 are abbreviated as l_1 and l_2 whereas the unlocking statements are abbreviated as u_1 and u_2, respectively. If l_1 (l_2) and u_1 (u_2) are added immediately before c_p ($d_{p'}$) and immediately after c_q ($d_{q'}$), respectively, then the resulting critical sections, i.e., the set of statements between (and including) l_1 (l_2) and u_1 (u_2) are denoted by $[\![c_p, c_q]\!]_l$ ($[\![d_{p'}, d_{q'}]\!]_l$).

During lock insertion, two sets of decisions need to be made:

- **Lock Statement Insertion**: determining locations of insertion of the lock statements l_1 and l_2, and
- **Unlock Statement Insertion**: determining locations of insertion of the matching unlock statements u_1 and u_2.

```
T1 () {
    ...
c0:   lock(m);
    ...
c1:   lock(n);
    ...
c2:   unlock(n);

    // begin critical section
    local1   = account_value;
    local1  += increment;
    account_value   = local1;
    // end critical section

c3:   unlock(m);
    ...
}
```

```
T2 () {
d0:   lock(n);
d1:   ...

    // begin critical section
    local2   = account_value;
d2:   lock(m);
    // access another account
    local2  += other_account_value;
d3:   unlock(m);
    account_value   = local2;
    // end critical section

d4:   unlock(n);
}
```

Fig. 2. Lock Insertion Example.

A. Insertion of Unlocking Statements

While determining locations where to insert the locking statements is not straightforward, we observe that since un-lock statements are non-blocking they cannot participate in a deadlock. It follows that in order to enforce the mutually atomicity of the segments $[c_i, c_j]$ and $[d_{i'}, d_{j'}]$, it suffices to insert the unlocking statements u_1 and u_2 immediately after c_j and $d_{j'}$, respectively. Formally,

Theorem 1 (Unlock Insertion). *Let* $[c_i, c_j]$ *and* $[d_{i'}, d_{j'}]$ *be segments of threads T_1 and T_2, respectively, defining mutually atomic segments to be enforced. Let $[\![c_a, c_b]\!]_l$ and $[\![d_{a'}, d_{b'}]\!]_l$, where $[c_i, c_j] \subseteq [\![c_a, c_b]\!]_l$ and $[d_{i'}, d_{j'}] \subseteq [\![d_{a'}, d_{b'}]\!]_l$, be critical sections enforcing mutually atomicity of $[c_i, c_j]$ and $[d_{i'}, d_{j'}]$ that also satisfy the lock insertion requirements. Then $c_b = c_j$ and $d_{b'} = d_{j'}$.*

B. Insertion of Locking Statements

We now turn to the more interesting problem of inserting the locking statements l_1 and l_2. If guaranteeing deadlock freedom were not a requirement then inserting the statements l_1 (l_2) immediately before locations c_i ($d_{i'}$) in thread T_1 (T_2), suffices. Clearly, the resulting critical sections $[\![c_i, c_j]\!]_l$ and $[\![d_{i'}, d_{j'}]\!]_l$ satisfy lock insertion requirement 1. Moreover, since by requirement 1, $[c_i, c_j]$ ($[d_{i'}, d_{j'}]$) must belong to any critical section enforced by our newly inserted lock/unlock statements in thread T_1 (T_2), we see that $[\![c_i, c_j]\!]_l$ and $[\![d_{i'}, d_{j'}]\!]_l$ would indeed be minimal (based on set inclusion) critical sections potentially satisfying the lock insertion requirements.

However, inserting l_1 and l_2 immediately before c_i and $d_{i'}$, respectively, could introduce new deadlocks. Consider, for example, the concurrent program $\mathcal{C}$ comprised of threads T_1 and T_2 with the desired critical sections shown in fig 2. The running of our lock insertion procedure is demonstrated on the CFGs of T_1 and T_2 in fig 3. Here the original lock/unlock statements have been shown as black circles while the mutually atomic segments (CS_1, CS_2) to be enforced as rectangles. Let $CS_1 = [c_i, c_j]$ and $CS_2 = [d_{i'}, d_{j'}]$. Inserting l_1 and l_2 (shown as white circles) immediately before c_i and $d_{i'}$ results in the threads shown in Fig. 3(a). Note that at location c_3 thread T_1 holds lock m which was acquired at c_0, whereas at location d_2, thread T_2 holds lock l_2 acquired at d_1. Thus at global control location (c_3, d_2) of $\mathcal{C}$, T_1 holds lock m and is waiting at acquire l, whereas T_2 holds l and is waiting to acquire m. This cyclic dependency creates a deadlock.

Thread T_1 move. Recall that our goal is to guarantee deadlock freedom while ensuring minimality of the newly introduced critical sections. Towards that end, we started by inserting the locking statements l_1 and l_2 immediately before c_i and $d_{i'}$, respectively, even if the newly synthesized threads have deadlocks. If no new deadlocks are introduced then we are done. If there exist newly introduced deadlocks, they must involve at least one of l_1 or l_2. In our case, $c_3 : l_1$ can potentially be involved in a deadlock but $d_1 : l_2$ cannot. Thus in order to guarantee deadlock freedom, we need to re-locate the locking statement l_1. We observe that l_1 cannot be moved forward as that would cause it to enter the critical section CS_1 which we are supposed to enforce. Thus l_1 can only be moved backwards along T_1.

In order to ensure that l_1 does not participate in a deadlock we move l_1 backwards along T_1 till we encounter a control location where it can no longer be involved in any deadlock. In order to identify this location, we recall that two conditions need to be satisfied in order for $c_k : l_1$ to be involved in a deadlock with a statement $d_{k'} : lock(m)$ of thread T_2.

1) **Reachability:** $(c_k, d_{k'})$ are pairwise reachable, and
2) **Cyclic Dependency:** locks m and l are held at c_k and $d_{k'}$, respectively

Thus in order to identify the location where to introduce l_1 in thread T_1, we keep moving it backwards starting from c_i till we encounter a control location c_k where at least one of the above conditions is falsified. By condition 2, we see that if $c_k : l_1$ deadlocks with location $d_{k'}$ of T_2, lock l must be held at $d_{k'}$. Thus it follows that the *lock(l)* statement in T_1 can deadlock only with a locking statement in the critical section $[l_2, u_2]$ in thread T_2. Motivated by the above observation, we define $L_{[l_2, u_2]}$ to be the set of locks p such that a statement of the form *lock(p)* occurs along $[l_2, u_2]$.

Let c_k, where $k \leq i$ be the last control location occurring before c_i along T_i such that (i) c_k violates condition 1 or 2, and (ii) for each $r \in [k+1..i]$, c_r does not violate any of the conditions 1 or 2. Then we insert l_1 immediately before c_k. Note that, by our construction, c_k is the first location encountered by traversing backwards along T_1 starting at c_i where a *lock(l)* statement can be inserted without it being involved in a deadlock. In our example, in order to remove all potential deadlocks involving l_1 we move it to location c_4 (see Fig. 3(b)).

Deadlock Check. Having removed the deadlocks involving l_1, we check whether l_2 is involved in a deadlock. If not then the procedure terminates.

Thread T_2 move. If, on the other hand, l_2 is involved in a deadlock we remove deadlocks involving l_2 using the same procedure as above - the only difference being that we now consider the deadlocks involving l_2 and the locks acquired along $[l_1, u_1]$. We keep moving l_2 backwards along T_2 till we reach a control location of T_2 where l_2 cannot be involved in a deadlock. In our example, we see that even though $d_1 : l_2$ couldn't be involved in a deadlock in the original program Fig. 3(a), in the new program Fig. 3(b) gotten via enlargement of the critical section induced by lock l in T_1, $d_1 : l_2$ can potentially deadlock with location c_1. In order to remove deadlocks involving l_2 we re-locate it back to location d_4.

Dovetailing. Note, however, that as we move l_2 backwards along T_2 we enlarge the critical section $[l_2, u_2]$. A key

consequence is that the enlarged critical section may contain new locking statements which may now induce new deadlocks with l_1. In order to remove these deadlocks we again repeat the above procedure by moving l_1 further backwards till it cannot be involved in a deadlock.

The whole process of removing deadlocks involving statements l_1 and l_2 in a dovetailed fashion, wherein the statements l_1 and l_2 are re-located backwards, is continued till all deadlocks involving l_1 and l_2 are removed. This yields us a deadlock free insertion of l_1 and l_2 in T_1 and T_2, respectively (see Fig. 3(c)).

A formal description of the lock insertion procedure is formulated as Alg. 1.

Algorithm 1 Lock Insertion for Straight-line Code

1: **Input:** Threads $T_1, ..., T_n$ specified as straight-line code, with T_1 and T_2 defined by the sequences $c_0, ..., c_n$ and $d_0, ..., d_m$, respectively, and mutually atomic segments $s_1 = [c_i, c_j]$ and $s_2 = [d_{i'}, d_{j'}]$ of T_1 and T_2, respectively.
2: Insert u_1 and u_2 in threads T_1 and T_2 immediately after c_j and $d_{j'}$, respectively. 　　　　(**Insertion of Unlock Statements**)
3: Insert l_1 and l_2 in threads T_1 and T_2 immediately before c_i and $d_{i'}$, respectively.
4: **repeat**
5: 　　**if** l_1 can be involved in a potential deadlock **then**
6: 　　　　Move l_1 backward along T_1 till we reach a control location c' of thread T_1 such that for each lock $m \in L_{[l_2, u_2]}$: either (i) m is not held at c', or (ii) for each location d' in critical section $[l_2, u_2]$ where m is acquired, c' and d' are not pairwise reachable.
7: 　　**end if**
8: 　　**if** l_2 can be involved in a potential deadlock **then**
9: 　　　　Move l_2 backward along T_2 till we reach a control location d' of thread T_2 such that for each lock $m \in L_{[l_1, u_1]}$: either (i) m is not held at d', or (ii) for each location c' in critical section $[l_1, u_1]$ where m is acquired, c' and d' are not pairwise reachable.
10: 　　**end if**
11: **until** there do not exist any deadlocks involving l_1 or l_2

C. Meeting Lock Insertion Requirements

We now show the somewhat surprising result that simply by making local moves of l_1 and l_2 in a dove-tailed manner as encoded in Alg. 1, all three of our (global) lock insertion requirements are met.

Enforcement of Mutual Atomicity. Since Alg. 1 always maintains the invariants that $[c_i, c_j] \subseteq [l_1, u_1]$ and $[d_{i'}, d_{j'}] \subseteq [l_2, u_2]$ we see that the mutual atomicity of $[c_i, c_j]$ and $[d_{i'}, d_{j'}]$ is enforced.

Deadlock Freedom. The termination condition (step 11) of Alg. 1 ensures that there do not exist deadlocks involving l_1 or l_2 and since the newly introduced deadlock must involve at least one of these lock statements we see that requirement 2 is also met.

Optimality. The most interesting part is to show that requirement 3 is met, i.e. the critical sections identified by Alg. 1
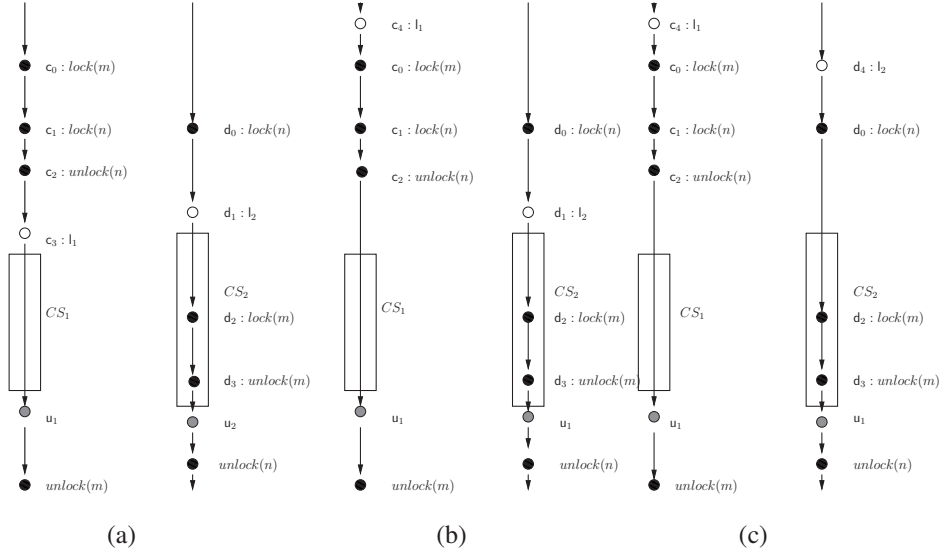
Fig. 3.   Lock Insertion Procedure

are optimal. The proof is provided in the full version of the paper [**?**].

Optimality Result. *Let $[c_a, c_b]$ and $[d_{a'}, d_{b'}]$ be critical sections satisfying the lock insertion requirements. Then $[l_1, u_1] \subseteq [c_a, c_b]$ and $[l_2, u_2] \subseteq [d_{a'}, d_{b'}]$, where $[l_1, u_1]$ and $[l_2, u_2]$ are the critical sections in threads T_1 and T_2, respectively, identified by Alg. 1.*

Proof.

We prove the result by contradiction. If possible, suppose that $[c_a, c_b]$ is a proper subset of $[l_1, u_1]$. As discussed before, the fact that unlock statements are non-blocking combined with the optimality requirement imply that $u_1 = c_b$ and $u_2 = d_{b'}$. Then from the assumption that $[c_a, c_b]$ is a proper subset of $[l_1, u_1]$ we can deduce that l_1 occurs before c_a along T_1.

For $r \geq 0$, let l_1^r and l_2^r be the locations of the *lock(l)* statements in threads T_1 and T_2 after the rth iteration of Alg. 1. Suppose that k is the largest index for which l_1^k belongs to the interval $[c_a, c_b]$ and l_2^k belongs to the interval $[c_{a'}, c_{b'}]$. At the $(k{+}1)$st iteration, either l_1 moves out of the interval $[c_a, c_b]$ or l_2 moves out of the interval $[c_{a'}, c_{b'}]$. For definiteness assume that it is the former.

To prove our claim we now show that l_1 cannot move out of the interval $[c_a, c_b]$. Indeed for it to move out, l_1 needs to be propagated backwards along T_1 till it crosses c_a. At the time of crossing c_a, T_1 must be holding a lock m such that (i) the last statement to acquire m occurs before c_a along T_1, and (ii) there exists a lock acquisition statement for m in the critical section $[l_2^k, u_2^k]$. Let L_{c_a} be the set of locks held at c_a that are also acquired in the critical section $[l_2^k, u_2^k]$. Clearly $L_{c_a} \neq \emptyset$. Furthermore, there exists a lock $m' \in L_{c_a}$ such that $(c_a, d_{m'})$ are pairwise reachable for some statement $d_{m'}$ acquiring lock m' in $[l_2^k, u_2^k]$. If that is not the case then l_1^{k+1} would not cross c_a contradicting the maximality of k. However this creates a deadlock involving locations c_a and $d_{m'}$ of T_1 and T_2, respectively. This is because at c_a, thread T_1 holds lock m' and is waiting to acquire lock l (recall that, by definition, c_a is a *lock(l)* statement) whereas at $d_{m'}$ thread T_2 is holding

lock l (as $d_{m'}$ lies in the critical section $[l_2^k, u_2^k]$) and waiting to acquire m'. Recall that $d_{m'} \in [l_2^k, u_2^k] \subseteq [d_{a'}, d_{b'}]$. Thus $[c_a, c_b]$ and $[d_{a'}, d_{b'}]$ do not meet lock insertion requirement 2 contradicting our hypothesis.

Similarly we may show that $[l_2, u_2] \subseteq [d_{a'}, d_{b'}]$.

Note that our notion of minimality for critical sections is based on set inclusion as opposed to the number of program statements comprising the critical section. This is the best one can hope for.

V. LOCK INSERTION: THE GENERAL CASE

Acyclic CFGs We start by considering the case where the CFGs of threads are acyclic. Here each atomic segment is a DAG (directed acyclic graph) with possibly multiple roots (vertices of in-degree zero) and possibly multiple leaves (vertices of out-degree zero). Thus we assume that the input to the procedure is a pair of mutually atomic segments $s_1 = [(c_{i1}, ..., c_{ik}), (c_{j1}, ..., c_{jp})]$ and $s_2 = [(d_{i'1}, ..., d_{i'k'}), (d_{j'1}, ..., d_{j'p'})]$, wherein the first (second) tuple in each segment represents the roots (leaves) of the segment. Generalization of Alg. 1 to DAGs requires little modification as the notion of backwards traversal required for steps 6 and 9 is well defined. The core idea of lock insertion remains the same as for straight-line code. We start by inserting the unlock statement u_1 immediately after the control locations $c_{j1}, ..., c_{jp}$ and the unlock statement u_2 immediately after the control locations $d_{j'1}, ..., d_{j'p'}$ (step 3 of Alg. 2). This step is analogous to the case of straight-line code, the only difference being that instead of a unique *leaf* node there are multiple *leaves*. Again, as for straight-line code, the locking statements l_1 and l_2 are inserted (step 4) immediately before the locations $c_{i1}, ..., c_{ik}$ and the locations $d_{i'1}, ..., d_{i'k'}$, respectively.

As before, in order to remove deadlocks involving l_1 and l_2 we propagate these statements upwards along the CFGs of the respective threads (steps 5-10 of Alg. 2) via Alg. 3.

In propagating the lock statements l_1 and l_2 upwards along the DAGs, there are two main differences from the straight-line case. First due to branching, we may encounter the same

control location multiple times. To track the control locations that have already been visited we maintain a set *Visited* and insert a check (step 10 of Alg. 3) to prevent repeat processing. Secondly, we may encounter control locations with multiple predecessors in which case we need to propagate the locking statement backwards along multiple branches (steps 9-17 of Alg. 3).

Cyclic CFGs. In the general case, due to the presence of cycles in the CFG (caused by loops) the notion of backwards traversal is not well-defined. However, we can reduce the problem of lock insertion for cyclic CFGs to the acyclic case. Towards that end, we leverage the consistency assumption (Sec. III) wherein if a loop head (tail) occurs in an atomic segment comprising a critical section then the matching loop tail (head) also occurs in some (possibly the same) segment comprising the same critical section.

In order to convert a cyclic CFG into an acyclic CFG we traverse the CFG CFG_i of thread T_i starting at its entry location in a depth-first manner and identify a set of back-edges. These back-edges transit from tails of loops to their matching heads. Deleting these back-edges results in an acyclic CFG which we denote by CFG_i'. Next we run Alg. 2 on CFG_i', the only difference being that if during the backward traversal of a newly introduced locking statement we include a loop tail lt for the first time in the critical section induced by the newly introduced locking statements, then in order to preserve the consistency invariant we also need to include the matching loop head in the critical section (as was discussed in Sec. 3). Thus if lh is the matching loop head for lt and if lh does not already exists in the an atomic segment in the current specification then we generate a new instance of the lock insertion problem by inserting the atomic segment comprising the loop head lh in the existing set of atomic segments (steps 12-15 of Alg. 3).

Algorithm 2 Lock Insertion for General Programs

1: **Input:** Threads T_1 and T_2 specified it terms of their respective CFGs and pairs of segments $s_1 = [s_1^1, s_1^2], ..., s_k = [s_k^1, s_k^2]$, where s_j^i is an atomic segment of T_i specified as a DAG that is a subgraph of the CFG CFG_i of T_i.
2: Assign a new lock l_i to segment pair s_i.
3: Insert unlock statements u_1^i and u_2^i for lock l_i in threads T_1 and T_2 immediately after the leaves of s_i^1 and s_i^2 in T_1 and T_2, respectively. (**Insertion of Unlock Statements**)
4: Insert lock statements l_1^i and l_2^i for lock l_i in threads T_1 and T_2 immediately before the roots of s_i^1 and s_i^2 in T_1 and T_2, respectively.
5: **for** each lock l_i, where $i \in [1..k]$ **do**
6:    **repeat**
7:       Remove deadlocks involving l_i^1 via Alg. 3
8:       Remove deadlocks involving l_i^2 via Alg. 3
9:    **until** there do not exist any deadlocks involving l_i^1 and l_i^2
10: **end for**

VI. IMPLEMENTATION

From Alg. 1, we see that the key step in lock insertion involves deciding, in an efficient manner, the multiple reach-

---

Algorithm 3 Remove Deadlocks

1: **Input:** Segments $s_{i1} = [s_{i1}^1, s_{i2}^2], ..., s_{ik_i} = [s_{ik_i}^1, s_{ik_i}^2]$, associated with the same lock l_i and thread T_m, where $m \in [1..2]$.
2: **Output:** Possible re-location of *lock(l)* statements in thread T_m in order to guarantee absence of deadlocks involving l_i^m, i.e., the *lock(l_i)* statement in T_m.
3: **for** each pair $s_{ij} = [s_{ij}^1, s_{ij}^2]$ **do**
4:    Set $Worklist$ to the locations of all the *lock(l_i)* statements enforcing s_{ij}^m in T_m that are involved in a deadlock
5:    $Visited = \emptyset$
6:    **while** $Worklist \neq \emptyset$ **do**
7:       Remove a location *loc* from *Worklist*
8:       **if** there exists a lock m held at *loc* that is acquired at a control location *loc'* in the segment $s_{ij}^{k'}$, where $k \neq k'$, and (loc, loc') are pairwise reachable **then**
9:          **for** each predecessor *pred* of *loc* **do**
10:             **if** $pred \notin Visited$ **then**
11:                Add *pred* to *Worklist* and to *Visited*.
12:                **if** *pred* is a loop tail *lt* that is not included in any of the segments s_{ir}^m, with $r \in [1..k_i]$ **then**
13:                   Construct a new segment *seg* comprising only of the loop head *lh* that matches *lt*
14:                   Add the new segment pair $sp = [sp^1, sp^2]$, where $sp^k = seg$ and $sp^{k'} = s_{ij}^{k'}$ with $k \neq k'$, and associate lock l_i with it
15:                **end if**
16:             **end if**
17:          **end for**
18:       **else**
19:          Insert *lock(l_i)* immediately before *loc*
20:       **end if**
21:    **end while**
22: **end for**

---

ability queries that are generated (via steps 6 and 9) as we traverse backwards along the CFGs of threads in the given program. However, in general, reachability of a pair of control locations in threads is not decidable. The strategy that is often used to bypass the decidability barrier is to consider reachability in the presence of synchronization primitives like locks and wait/notify only and ignore data variables. This is referred to as *static reachability*. We observe that relying on static reachability instead of reachability guarantees soundness of our procedure.

A. Nested Locks

Alg. 1 formulates an optimal procedure for lock insertion for concurrent programs with arbitrary locking patterns. However, in real world applications most lock usage is *nested* [10], where we say that a concurrent program accesses locks in a *nested* fashion 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. This has two main implications. First, it is known that while static reachability is undecidable for arbitrary locking patterns it not

only becomes decidable for nested locks but efficiently so [10]. Thus a key advantage of nestedness is that it enable us to leverage efficient procedures for deciding static reachability thereby yielding a fast and effective lock insertion procedure.

Secondly, if a program has nested locks to start with, we would like to preserve this nestedness. Our general lock insertion procedure, however, may violate nestedness. We therefore formulate a modified procedure that ensures that nestedness is preserved in the newly synthesized program. Note that this procedure still guarantees optimality of newly introduced critical sections if we restrict ourselves to the space of programs with nested locks only.

B. Review of Acquisition Histories

We start by reviewing the notion of acquisition histories [10] that have been used for efficiently reasoning about static reachability for nested locks.

Definition (Acquisition History) *For a lock l held by thread T at a control location d, the acquisition history of l along a local computation x of T leading to c, denoted by $\mathsf{ah}_T(c, l, x)$, is the set of locks that have been acquired (and possibly released) by T since the last acquisition of l by T in traversing forward along x to c.*

Acquisition histories enable us to formulate a necessary and sufficient condition for static reachability for nested locks.

Theorem 2 (Decomposition Result) [9]. *Let x^i be a local computation of T_i leading to c_i. Then (c_1, c_2) is statically reachable via an interleaving of x^1 and x^2 if and only if (i) the locks held at c_1 and c_2 are disjoint, and (ii) the acquisition histories at c_1 and c_2 are consistent, i.e., there do not exist locks l and l' that are held at c_1 and c_2, respectively, such that $l \in \mathsf{ah}_{T_2}(c_2, l', x^2)$ and $l' \in \mathsf{ah}_{T_1}(c_1, l, x^1)$.*

The reason we refer to thm. 2 as the decomposition result is that it enables us to reason about static reachability for nested locks in a thread local manner. This is because much like locksets, acquisition histories can be computed *thread locally* at each location of interest in thread T via a simple traversal of the CFG of T. This is key to ensuring efficiency of deciding static reachability for nested locks.

Let $\mathsf{AH}_{T_i}(c_i)$ be the set of all possible acquisition histories encountered along paths of T_i leading to c_i. Then from thm. 2, we have the following criterion for static reachability between global control states.

Corollary (Generalized Decomposition Result). *Global control states $\mathsf{c} = (c_1, c_2)$ is statically reachable if and only if (1) disjoint sets of locks are held at c_1 and c_2, and, (ii) there exist acquisition histories $\mathsf{ah}_1 \in AH_{T_1}(c_1)$ and $\mathsf{ah}_2 \in AH_{T_2}(c_2)$ that are consistent.*

Nested Lock Insertion. In applying the decomposition result during lock insertion we face two main challenges. First, as we traverse the CFGs of threads backwards, we generate multiple (static) pairwise reachability queries. Thus we want to avoid computing acquisition histories between the same pair of control locations multiple times. Towards that end, we pre-compute, in one pre-processing step, the acquisition histories at all *relevant* control locations of interest in each thread.

Localizing the Analysis. The key issue next is how to localize these locations of interest. Towards that end, let LF_i be the set of last lock free (where no lock is held) locations along local paths of T_i leading to an entry location of the critical section to be enforced. Note that we can simply insert the *lock(l)* statement immediately after locations in LF_1 and LF_2. This ensures that desired critical sections are enforced and no new deadlocks are introduced as no lock is held at any of the locations in LF_1 or LF_2. However, this may not lead to optimal critical sections. It follows that, in order to achieve optimality, our lock insertion strategy can only introduce *lock(l)* statements immediately before an existing locking statement occurring along paths from locations in LF_i to the entry locations of the desired critical section. We call the set of all such locking statements *History Lock Statements* as they are in the acquisition history of the critical section that we are trying to enforce. Thus it suffices to compute $\mathsf{AH}_{T_i}(c_i)$ only for history lock statements c_i of T_i.

Once the acquisition histories have been computed, the procedure for lock insertion for the nested case can then be formulated as Alg. 4. Note that the main difference between algs. 1 and 4 is that pairwise reachability is determined using acquisition histories computed in steps 2-4. Alg. 2 for the general case can also be modified accordingly.

Algorithm 4 Nested Lock Insertion via Acquisition Histories

1: **Input:** Threads T_1 and T_2 specified as control flow graphs, CFG_1 and CFG_2, respectively, and mutually atomic segments $s_1 = [c_i, c_j]$ and $s_2 = [d_{i'}, d_{j'}]$ of T_1 and T_2, respectively.
2: **for** each thread T_i **do**
3:     Compute the lock acquisition histories $\mathsf{AH}_{T_i}(c)$ at each location c where c is a history lock statement of C_i in T_i
4: **end for**
5: Insert l_1 and l_2 in threads T_1 and T_2 immediately before c_i and $d_{i'}$, respectively.
6: **repeat**
7:     **if** l_1 can be involved in a potential deadlock **then**
8:         Move l_1 backward along T_1 via a backward DFS traversal of CFG_i till we reach control locations c' of thread T_1 such that for each lock $m \in L_{[\mathsf{l}_2, \mathsf{u}_2]}$: either (i) m is not held at c', or (ii) for each location d' in critical section $[\mathsf{l}_2, \mathsf{u}_2]$ where m is acquired, $\mathsf{AH}_{T_i}(c')$ and $\mathsf{AH}_{T_i}(d')$ are not consistent.
9:     **end if**
10:     **if** l_2 can be involved in a potential deadlock **then**
11:         Move l_2 backward along T_2 till we reach a control location d' of thread T_2 such that for each lock $m \in L_{[\mathsf{l}_1, \mathsf{u}_1]}$: either (i) m is not held at d', or (ii) for each location c' in critical section $[\mathsf{l}_1, \mathsf{u}_1]$ where m is acquired, $\mathsf{AH}_{T_1}(c')$ and $\mathsf{AH}_{T_2}(d')$ are not consistent.
12:     **end if**
13: **until** there do not exist any potential deadlocks involving l_1 or l_2
14: Add unlock statement to match l_1 and l_2 in a manner that ensure that locks are nested.

Example	KLOC	Segment Pairs	Acquisition History Computation	Lock Insertion (secs)
account.Main	50 LOC	2	0.9	0.1
atom001a	70 LOC	3	1.4	0.2
atom002a	75 LOC	3	1.6	0.3
banking-av	150 LOC	1	1.1	0.3
banking-sav	175 LOC	2	1.2	0.4
D-1	2.9	3	1.2	0.4
D-2	8.3	12	7.4	1.1
D-3	8.3	3	8	1.2
D-4	17.8	9	6.7	4.4
D-5	17.8	2	7	2.5

TABLE I
LOCK INSERTION DATA

C. Guaranteeing Nestedness of Locks

Alg. 4 does not guarantee preservation of nestedness of locks. To ensure nestedness we make two modifications to Alg. 4. First, instead of inserting the *unlock(l)* statement before the *lock(l)* statement, we first insert the *lock(l)* statement and then add the matching *unlock(l)* statements to ensure nestedness of locks. However, we need to make sure that the *lock(l)* statements are inserted at locations such that there exist locations where the matching *unlock(l)* statements can be inserted to ensure nestedness. This is accomplished by augmenting the conditions in steps 8 and 11 with the extra constraint that the matching unlock statements *unlock(l)* can be inserted so as to enforce the desired critical sections while preserving nestedness.

VII. EXPERIMENTS

We consider a set of public benchmarks with known atomicity violations used in our previous work [11]. These are small examples and are used mainly to illustrate the efficacy of our new lock insertion technique. We also use an in-house parallel implementation of an MPEG-4 decoder **S** with known atomicity violations detected via static and run-time techniques. Finally we also consider a large in-house concurrent software system implementing a distributed storage system, denoted by **D**. The **D** system consists of about 400K lines of C++ code using Boost libraries and is based on a thread pool model. We evaluated our approach by applying our technique to different modules of **D** denoted by **D**-1, **D**-2, **D**-3, **D**-4 and **D**-5.

We present the time taken for the context-sensitive points-to analysis for the lock pointers and the pre-processing step that computes the acquisition histories at locations of interest (col. 4) and the time taken for the lock insertion procedure (col. 5). The key thing worth noting is that the lock insertion procedure is efficient even for large examples (col. 5). In fact the total time taken is dominated by the points-to analysis and the acquisition history computation. This is to be expected as once the acquisition histories have been computed the lock insertion procedure involves highly localized dovetailed movements of the lock statements around critical sections to be enforced. Usually these movements are restricted to function boundaries. On the other hand, computing the points-to sets requires us to reason about code modules that may impact aliases of relevant lock pointers at locations of interest as opposed to just a few functions where the atomicity violations need to be fixed.

VIII. RELATED WORK AND CONCLUSION

There has been interesting work on automatically inferring locks for atomic sections [4], [12], [6], [1], [2]. However most of this work has focused on allocating/inferring locks for programs with no prior locks. The absence of locks allows one greater control over lock placement thereby making it easier to enforce the desired correctness properties. For instance, deadlocks can be prevented simply by allocating locks in a fixed global order. One does not have this freedom if locks are required to be inserted in a program with existing locks. This makes the problem of lock insertion more challenging than lock allocation/inference. There is also limited amount of work on exploiting program semantics to insert synchronization statements in order to fix bugs [5], enforce concurrency control in order to satisfy invariants [3], or ensure correctness [13]. However reasoning about program semantics requires the use of refined heavy-weight analyses like constraint/SAT solving or state space exploration via model checking.

In contrast, we have formulated a fully automatic, provably optimal, efficient and precise technique for lock insertion in concurrent code with pre-existing locks that ensures deadlock freedom while attempting to minimize the resulting critical sections. Importantly, our method localizes the analysis to only the necessary code modules. Moreover, for the special case of programs with nested locks our analysis is compositional, i.e., thread local, thereby avoiding a global analysis and ensuring scalability to large real-life programs.

REFERENCES

[1] Sigmund Cherem, Trishul M. Chilimbi, and Sumit Gulwani. Inferring locks for atomic sections. In *PLDI*, 2008.
[2] Dave Cunningham, Khilan Gudka, and Susan Eisenbach. Keep off the grass: Locking the right path for atomicity. In *CC*, 2008.
[3] Jyotirmoy V. Deshmukh, G. Ramalingam, Venkatesh Prasad Ranganath, and Kapil Vaswani. Logical concurrency control from sequential proofs. In *ESOP*, 2010.
[4] M. Emmi, J. S. Fischer, R. Jhala, and R. Majumdar. Lock allocation. In *POPL*, 2007.
[5] C. Flanagan and S. N. Freund. Automatic synchronization correction. In *SCOOL*, 2005.
[6] M. Hicks, J. S. Foster, and P. Pratikakis. Lock inference for atomic sections. In *First Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*.
[7] Takashi Horikawa. An approach for scalability-bottleneck solution: identification and elimination of scalability bottlenecks in a dbms. In *ICPE*, 2011.
[8] Takashi Horikawa. An approach for scalability-bottleneck solution: identification and elimination of scalability bottlenecks in a dbms (abstracts only). *SIGMETRICS Performance Evaluation Review*, 39(3), 2011.
[9] V. Kahlon and A. Gupta. On the Analysis of Interacting Pushdown Systems. In *POPL*, 2007.
[10] V. Kahlon, F. Ivančić, and A. Gupta. Reasoning about threads communicating via locks. In *CAV*, 2005.
[11] V. Kahlon and C. Wang. Universal causality graphs: A precise happens-before model for detecting bugs in concurrent programs. In *CAV*, 2010.
[12] Bill McCloskey, Feng Zhou, David Gay, and Eric A. Brewer. Autolocker: synchronization inference for atomic sections. In *POPL*, 2006.
[13] M. Vechev, E. Yahav, and G. Yorsh. Inferring synchronization under limited observability. In *TACAS*, 2009.