Identifying and Validating Irregular Mutual Exclusion Synchronization in Explicitly Parallel Programs

Diego Novillo[1], Ronald C. Unrau[1], and Jonathan Schaeffer[2]

[1] Red Hat Inc., Sunnyvale, CA 94089, USA
{dnovillo,runrau}@redhat.com

[2] Computing Science Department, University of Alberta, Edmonton, Alberta, Canada T6G 2H1
jonathan@cs.ualberta.ca

Abstract. Existing work on mutual exclusion synchronization is based on a structural definition of mutex bodies. Although correct, this structural notion fails to identify many important locking patterns present in some programs. In this paper we present a novel analysis technique for identifying mutual exclusion synchronization patterns in explicitly parallel programs. We use this analysis in a new technique, called *lock-picking*, which detects and eliminates redundant mutex operations. We also show that this new mutex analysis technique can be used as a validation tool in a compiler. Using this analysis, a compiler can detect irregularities like lock tripping, deadlock patterns, incomplete mutex bodies, dangling `lock` and `unlock` operations and partially protected code.

1 Introduction

In this paper we present a novel analysis technique for identifying mutual exclusion synchronization patterns in explicitly parallel programs. We apply this analysis to develop a new technique, called *lock-picking*, to detect and eliminate redundant mutex operations. We also show that this new mutex analysis technique can be used as a validation tool in a compiler. We build on a concurrent data-flow analysis framework called CSSAME (Concurrent Static Single Assignment with Mutual Exclusion,pronounced *sesame*) [6] to analyze and optimize the synchronization framework of both task and data parallel programs. We have implemented these algorithms and apply them to several concurrent and sequential applications.

2 The CSSAME Form

The CSSAME form is a refinement of the Concurrent SSA (CSSA) framework [3] that incorporates mutual exclusion synchronization analysis to identify memory interleavings that are not possible at runtime due to the synchronization structure of the program. CSSAME extends CSSA to include mutual exclusion synchronization and barrier synchronization [5].

Like the sequential SSA form, CSSAME has the property that every use of a variable is reached by exactly one definition. Two merge operators are used in the CSSAME form: ϕ functions and π functions. A ϕ function merges all the incoming control reaching definitions to create a new definition for the variable. Control reaching definitions are those that reach a use u via sequential flow of execution (i.e., the definition has been made by the same thread). The second merge operator is the π function, which merges concurrent reaching definitions. Concurrent reaching definitions are those that reach a use u from other threads.

3 Motivation and Overview

Given an arbitrary statement s in a program and a lock variable L, a mutex structure analyzer should be able to answer the question "does s execute under the protection of lock L?". The answer to that question should be one of *always*, *never* or *sometimes*. To be conservatively correct, the compiler treats *never* and *sometimes* as equivalent. Furthermore, if the analysis determines that statement s is sometimes protected and sometimes not, this information could be used to warn the user about an anomalous locking pattern.

Existing work on mutual exclusion synchronization is based on a structural definition of mutex bodies [2, 4, 6]. A mutex body is indicated by a pair of `lock` and `unlock` nodes. All the graph nodes dominated by the `lock` node and post-dominated by the `unlock` node are part of the mutex body. Although correct, this notion of mutex body fails to identify some valid locking patterns present in some programs.

For example, consider the code fragment in Figure 1, which is part of a quicksort algorithm taken from the TreadMarks DSM system. We are interested in the mutual exclusion sections created by the lock variable *TSL*. Notice that a structural definition of mutex bodies will identify no mutex bodies in this function. The only `lock`/`unlock` pair that might qualify as a mutex body are the statements L_1 and U_3 (lines 3 and 37 respectively). However, the presence of other `lock` and `unlock` operations in between these statements forces the compiler to disregard this pair as a valid mutex body. A closer inspection reveals that the only statement that executes without lock protection is the busy wait statement S_1 (line 24).

4 Detecting mutex structures

A mutex structure for lock variable L is the set of all the mutex bodies for L in the program. To detect mutex structures, the intermediate representation for the program is modified so that (a) every graph node contains a use for *each* lock variable in the program, and, (b) for each lock variable L the graph entry node is assumed to contain an `unlock(L)` operation (i.e., variables are initially "unlocked").

Mutex structures are detected using sequential reaching definition information for each lock variable L. Nodes that are only reached by definitions of L coming from `lock(L)` nodes are protected by L. Nodes that can be reached by at least one `unlock(L)` node are not protected by L. Using this information we build an initial set of mutex bodies for each individual `lock(L)` node in the graph. This initial set is then refined by merging mutex bodies with common nodes [5]. This mutex analysis framework can be used as a validation tool in a compiler. Using this analysis, a compiler can detect irregularities like [5]:

Lock Tripping. Let L be a lock variable and n be a `lock(L)` node. Suppose that n is reached by other `lock(L)` nodes. If all the definitions come from other `lock(L)`

```
int PopWork(TaskElement *task)
{
  L1 ⇒ lock(TSL);
  while (TaskStackTop == 0) {
    if (++NumWaiting == NPROCS) {
      /* All the threads are waiting for work.
       * We are done.
       */
      lock(pause_lock);
      pause_flag = 1;
      unlock(pause_lock);
      U1 ⇒ unlock(TSL);
      return DONE;
    } else {
      if (NumWaiting == 1) {
        lock(pause_lock);
        pause_flag = 0;
        unlock(pause_lock);
      }
      U2 ⇒ unlock(TSL);
      /* Wait for work. This is the only
       * statement not protected by TSL.
       */
      S1 ⇒ while (!pause_flag) ; /* busy-wait */
      L2 ⇒ lock(TSL);
      if (NumWaiting == NPROCS) {
        U3 ⇒ unlock(TSL);
        return DONE;
      }
      --NumWaiting;
    } /* endif ++NumWorking == NPROCS */
  } /* while task-stack empty */
  /* Pop a piece of work from the stack */
  TaskStackTop--;
  task->left  = TaskStack[TaskStackTop].left;
  task->right = TaskStack[TaskStackTop].right;
  U3 ⇒ unlock(TSL);
  return 0;
}
```

Fig. 1. Locking pattern in function *PopWork()*.

nodes, the program is guaranteed to trip over lock L at runtime. If only some definitions come from other `lock(L)` nodes, the program may or may not trip over lock L.

Deadlock. Let L and M be two different lock variables such that in thread T_1 there is a `lock(L)` node that reaches a `lock(M)` node. In another thread T_2 a `lock(M)` node reaches a `lock(L)` node. If both T_1 and T_2 can execute concurrently, then the program may deadlock at runtime.

Incomplete mutex bodies. Let $B_L(n)$ be a partially built mutex body for L such that no node in $B_L(n)$ is an `unlock(L)` node. At runtime, if lock L is acquired at n, it will not be released.

Dangling unlock operations. Let x be an unlock node for L such that the set of reaching definitions for L at x does not include a `lock(L)` node. This indicates that the calling thread is releasing a lock that it has not acquired.

5 Lock-picking

Sometimes it is possible to remove synchronization operations from a program without affecting its semantics. For example, mutual exclusion synchronization is unnecessary in a sequential program and can be safely removed. In this section we describe *lock-picking*, a transformation that finds and removes superfluous `lock` and `unlock` operations. We say that a mutex body can be *lock-picked* if its lock and unlock nodes can be removed. An important property of lock picking is that it *does not* need to examine the mutex bodies of the program. Only the lock and unlock nodes are analyzed.

The lock-picking algorithm [5] examines the lock nodes for every mutex body in the program. The decision to lock-pick a mutex body is based on the absence of π functions for one or more lock variables at each mutex body lock node. The absence of π functions for lock variables at lock nodes means that there are no concurrent threads trying

```
double Sum = 0;
parloop (p, 0, N) {
  ...
  for (i = 0; i < M; i++) {
    S_3 = π(S_0, S_1, S_2);
    R_3 = π(R_0, R_1, R_2);
    lock(R_1);
    for (j = 0; j < M; j++) {
      sum_reduction(A[i][j]);
    }
    unlock(R_2);
  }
  ...
}

sum_reduction(double x)
{
  S_4 = π(S_0, S_1, S_2)
  R_4 = π(R_0, R_1, R_2)
  lock(S_1);
  Sum = Sum + x;
  unlock(S_2);
}
```

(a) Original CSSAME form.

```
double Sum = 0;
parloop (p, 0, N) {
  ...
  for (i = 0; i < M; i++) {
    S_3 = π(S_0, S_1, S_2)
    R_3 = π(R_0, R_1, R_2)
    lock(R_1);
    for (j = 0; j < M; j++) {
      S_4 = π(S_0, S_1, S_2)
      lock(S_1);
      Sum = Sum + A[i][j];
      unlock(S_2);
    }
    unlock(R_2);
  }
  ...
}
```

(b) CSSAME form after inlining and π pruning.

```
double Sum = 0;
parloop (p, 0, N) {
  ...
  for (i = 0; i < M; i++) {
    R_3 = π(R_0, R_1, R_2)
    lock(R_1);
    for (j = 0; j < M; j++) {
      Sum = Sum + A[i][j];
    }
    unlock(R_2);
  }
  ...
}
```

(c) After lock-picking.

Fig. 2. Effects of lock-picking on nested mutex bodies.

to acquire that lock. These conditions are typically discovered using whole program analysis. For example, consider the program in Figure 2(a). The inner loop calls the function *sum_reduction* to update a global reduction variable. Since *sum_reduction* is a generic reduction function, it locks the variable before doing the reduction. However, as a result of inlining, reduction lock S is no longer necessary because the reduction is always protected by lock R (Figure 2(b)). When *sum_reduction* is inlined, the use of R at the lock node for S becomes a protected use and its π function can be removed [6] (Figure 2(c)). In this case we say that the mutex structure for lock S is *nested* inside the mutex structure for R.

6 Experimental Results

We selected programs originally written in Java because we anticipated optimization opportunities due to the thread-safe nature of its libraries. Since Java libraries are thread-safe, application programs may spend up to half their execution time performing unnecessary synchronization [1]. The key reason for this overhead is that the libraries are generic and are not specific to an individual application's context. Hence, they have to be conservative in the assumptions they make. Therefore, when considered within the context of an actual program it might turn out that most of the synchronization operations are not necessary.

Table 1 shows the improvements obtained by applying lock-picking to sequential Java programs found in the JGL abstract class library these programs. We executed both the Java and C versions of these programs; in both cases the results were similar.

Benchmark	Unoptimized time (secs)	Optimized time (secs)	Relative Speedup
Array (1,000)	23	20	1.15
Array (10,000)	547	534	1.02
Map (3,000)	32	30	1.07
Map (30,000)	273	227	1.20
Sort (3,000)	32	30	1.07
Sort (30,000)	407	327	1.24

Table 1. Effect of lock-picking (LP) on sequential Java programs.

In general, we obtained performance improvements between 2% and 24% when lock-picking was applied. The performance gains obtained by removing the unnecessary locks are directly related to this particular implementation of mutual exclusion. Since these are sequential programs, all the synchronization overhead is caused by the actual call to `lock` and `unlock`. There is no lock contention. An alternative to removing the locks would have been to use a more efficient mutual exclusion synchronization implementation. We are convinced that a combination of compiler optimizations and efficient lock implementations is the best approach in these cases.

7 Conclusions

Synchronization analysis techniques are important in the context of an optimizing compiler for explicitly parallel programs. By reducing the number of memory conflicts, they simplify subsequent analysis and allow more aggressive optimizations to be applied.

In this paper we have developed a new technique to analyze non-concurrency for mutex synchronization that can handle locking patterns not supported by existing techniques. This allows the analysis of more complex mutual exclusion synchronization patterns in explicitly parallel programs. We have shown that this analysis can help detect common locking irregularities in parallel programs. Finally, we apply this analysis to remove mutex synchronization when it can be proven superfluous.

References

[1] D. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin Locks: Featherweight Synchronization for Java. In *ACM SIGPLAN '98*, June 1998.

[2] A. Krishnamurthy and K. Yelick. Analyses and Optimizations for Shared Address Space Programs. *J. Parallel and Distributed Computing*, 38:130–144, 1996.

[3] J. Lee, S. Midkiff, and D. A. Padua. Concurrent static single assignment form and constant propagation for explicitly parallel programs. In *LCPC '97*, August 1997.

[4] S. P. Masticola. *Static Detection of Deadlocks in Polynomial Time*. PhD thesis, Department of Computer Science, Rutgers University, 1993.

[5] D. Novillo. *Compiler Analysis and Optimization Techniques for Explicitly Parallel Programs*. PhD thesis, University of Alberta, February 2000.

[6] D. Novillo, R. Unrau, and J. Schaeffer. Concurrent SSA Form in the Presence of Mutual Exclusion. In *ICPP '98*, pages 356–364, August 1998.