Optimizing Mutual Exclusion Synchronization in Explicitly Parallel Programs

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

[1] Red Hat Canada, Ltd., Toronto, Ontario, Canada
{dnovillo,runrau}@redhat.com

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

Abstract. We present two new compiler optimizations for explicitly parallel programs based on the CSSAME form: Lock-Independent Code Motion (LICM) and Mutex Body Localization (MBL). We have implemented these optimizations on top of the SUIF framework and present performance results for selected SPLASH applications.

1 Introduction

Optimizations for explicitly parallel programs fall into two classes: the adaptation of sequential optimizations to a parallel environment; and the direct optimization of the parallel structure of the program. There have been several recent advances in adapting sequential optimizations such as global constant propagation and dead-code elimination to explicitly parallel programs [6, 10, 13]. There has been less emphasis on optimizing the parallel structure of the program itself.

We build on a concurrent dataflow analysis framework called CSSAME[1][12] to analyze and optimize the synchronization structure of both task and data parallel programs. *Lock-Independent Code Motion* (LICM) is an optimizing transformation that can reduce the size of critical sections in the program. *Mutex Body Localization* (MBL) is a new transformation that converts references to shared memory into references to local memory inside critical sections of the code. We have implemented these algorithms on top of the SUIF framework [5] and apply them to two SPLASH applications [15]: Water and Ocean. We also show that our techniques can be used to automate common optimizations that experienced programmers are currently forced to perform manually.

2 Related Work

Previous work in the area of optimizing explicitly parallel programs has concentrated on the adaptation of sequential optimization techniques to the parallel case. Lee, Midkiff and Padua propose a Concurrent SSA framework (CSSA) for explicitly parallel programs and interleaving memory semantics [9]. They adapt a constant propagation

[1] Concurrent Static Single Assignment with Mutual Exclusion. Pronounced *sesame*.

algorithm using the CSSA form. In recent work they have also adapted other SSA-based techniques including common subexpression elimination and code hoisting [10]. Their work only considers event-based synchronization operations and imposes some restrictions on the input program. Knoop, Steffen and Vollmer developed a bitvector analysis framework for parallel programs with shared memory and interleaving semantics [7]. They use their framework to adapt lazy code motion. However, their framework does not include synchronization operations. This reduces the opportunities for optimization in the general case.

In previous work we have extended the CSSA framework to incorporate mutual exclusion synchronization [13]. Our work extends the analysis techniques proposed by Lee *et al.* and shows the benefit of these extensions in the context of constant propagation for explicitly parallel programs. We also adapt a sequential dead-code removal algorithm that takes advantage of mutual exclusion information and describe an earlier form of the LICM technique that we extend and improve in this paper.

3 The CSSAME Form

The CSSAME form is a refinement of the CSSA framework [9] that incorporates more synchronization analysis to identify memory interleavings that are not possible at runtime due to the synchronization structure of the program. While CSSA only recognizes event synchronization, CSSAME extends it to include mutual exclusion synchronization and barrier synchronization [12]. CSSAME can be applied to both task and data parallel programs.

Like the sequential SSA form, CSSAME has the property that every use of a variable is reached by exactly one definition. When the flow of control causes more than one definition to reach a particular use, merge operators are introduced to resolve the ambiguity. Two merge operators are used in the CSSAME form: ϕ functions and π functions. ϕ functions have the same meaning as in sequential SSA [4]. π functions merge concurrent reaching definitions. Concurrent reaching definitions are those that reach the use of a variable from other threads.

The CSSAME form also examines π functions in critical sections of the code to determine if they can be removed. Since these π functions are in serialized sections of the code, some conflicts inside mutex bodies become superfluous and can be discarded. This pruning process is based on two observations:

1. *consecutive kills*: only definitions that reach the exit points of a critical section can be observed by other threads.
2. *protected uses*: if the use of a variable is protected by a definition local to the critical section, then definitions coming from concurrent critical sections will not be observed in this thread.

These two conditions allow the removal of superfluous conflict edges which in turn may lead to the complete removal of π functions, thus creating more opportunities for optimization [12]. The mutex synchronization analyzer modifies every node in the flowgraph so that they contain a use for each lock variable L (`lock` and `unlock` nodes already contain a definition and a use for L). To determine whether or not a flow graph node n is protected by lock L we compute reaching definition information for the use of L at n. If at least one of the reaching definitions comes from an `unlock` node or if there are no reaching definitions, then node n is not protected by lock L [12].

Mutex bodies are defined in terms of lock-protected nodes. In general, a mutex body $B_L(N)$ for lock variable L is a multiple-entry, multiple-exit region of the graph that encompasses all the flowgraph nodes that are protected by a common set of `lock` nodes (N). In contrast, previous work [8, 11] has treated mutex bodies as single-entry, single-exit regions. A mutex structure for a lock variable L is the set of all the mutex bodies for L in the program.

4 Lock-Independent Code Motion

Lock-Independent Code Motion (LICM) is a code motion technique that attempts to minimize the amount of code executed inside a mutex body. This optimization analyzes each mutex body to find code that can be moved outside. If at the end of the transformation a mutex body only contains unlock nodes, then the `lock` and `unlock` instructions are removed.

An expression E inside a mutex body $B_L(N)$ is lock-independent with respect to L if moving E outside $B_L(N)$ does not change the meaning of the program. Similarly, a statement (or group of statements) s is lock independent with respect to L if all the expressions and definitions in s are lock-independent. A flowgraph node n is lock independent if all its statements are lock-independent. The concept of lock-independence is similar to the concept of loop-invariant code for standard loop optimization techniques. Loop invariant computations compute the same result whether they are inside the loop or not. Analogously, lock-independent code computes the same result whether it is inside a mutex body or not. For instance, a statement s that references variables private to the thread will compute the same value whether it is executed inside a mutex body or not. This is also true if s references globally shared variables not modified by any other thread concurrent with s.

Lock-independent code is moved to special nodes called *premutex* and *postmutex* nodes. For every mutex body $B_L(N)$ there is a premutex node, denoted $premutex(n_i)$, for each `lock` node $n_i \in N$. Each premutex node $premutex(n_i)$ immediate dominates its associated `lock` node n_i. Similarly, there is a postmutex node, denoted $postmutex(x_i)$ for every `unlock` node x_i. Postmutex nodes are created as immediate post-dominators of each exit node x_i.

4.1 Moving Statements to Premutex Nodes

Given a lock-independent statement s inside a mutex body $B_L(N)$, LICM will attempt to move s to premutex or postmutex nodes for $B_L(N)$. The selection of `lock` nodes to receive statement s in their premutex node is done satisfying the following conditions (proofs of correctness are available separately [12]):

- **Protection.** Candidate `lock` nodes are initially selected among all the `lock` nodes in N that reach the node containing s (denoted $node(s)$). This condition provides an initial set of candidate `lock` nodes called $protectors(s)$.

Reachability. Since s is reached by all the nodes in *protectors*(s), there is a control path between each `lock` node in *protectors*(s) and *node*(s). Therefore, when statement s is removed from its original location, the statement must be replaced on every path from each `lock` node to *node*(s). This implies that s may need to be replicated to more than one premutex node.

To determine which `lock` nodes could receive a copy of s we perform reachability analysis among the `lock` nodes reaching s (*protectors*(s)). This analysis computes a partition of *protectors*(s), called *receivers*(s), that contains all the `lock` nodes that may receive a copy of statement s. The selection of receiver nodes is done so that (a) there exists a path between s and every `lock` node in *protectors*(s), and (b) instances of s occur only once along any of these paths (i.e., s is not unnecessarily replicated).

Algorithm 1 computes all the different sets of `lock` nodes that may receive a lock-independent statement s in their premutex nodes. Basically, the algorithm computes reachability sets among the nodes in *protectors*(s). The set *protectors*(s) is partitioned into k partitions $P_1, P_2, \ldots P_k$. Nodes in each partition P_j cannot reach each other but put together they reach or are reached by every other node in *protectors*(s). These partitions are the sets of `lock` nodes that can receive a copy of s in their premutex nodes.

Data Dependencies. When moving a statement s to one of the receiver sets for s, the motion must not alter the original data dependencies for the statement and other statements in the program. If P_j is the selected receiver set for s, two restrictions must be observed:

1. No variable defined by s may be used or defined along any path from *node*(s) to every node in P_j.
2. No variable used by s may be defined along any path from *node*(s) to every node in P_j.

These two restrictions are used to prune the set of receiver nodes computed in Algorithm 1. Notice that since the program is in CSSAME form, ϕ functions are also considered definitions and uses for a variable.

When more than one statement is moved to the same premutex node, the original data dependencies among the statements in the same premutex node must also be preserved. This is accomplished by maintaining the original control precedence when moving statements into the premutex node.

It is also possible to move statements forward to postmutex nodes of a mutex body $B_L(N)$. The analysis for postmutex nodes is similar to the previous case. The conditions are essentially the reverse of the conditions required for premutex nodes [12].

The LICMS algorithm scans all the mutex bodies in the program looking for lock-independent statements to move outside the mutex body. Each lock-independent statement s is checked against the conditions described previously. Lines $8-15$ in Algorithm 2 determine the sets of premutex receivers for s. The initial set of candidates computed by Algorithm 1 checks every lock node in a mutex body against each other looking for paths between them.

Notice that it might be possible that a statement can be moved to both the premutex and the postmutex nodes. In that case a cost model should determine which node is more

Algorithm 1 Compute candidate premutex nodes (*receivers*).

INPUT: A mutex body $B_L(N)$ and a lock-independent statement s.
OUTPUT: A list of receiver sets. Each receiver set P_i contains `lock` nodes whose premutex nodes may receive s.

$protectors(s) \leftarrow$ set of `lock` nodes that reach s.
$Q \leftarrow protectors(s)$
$k \leftarrow 1$
while $Q \neq \emptyset$ **do**
  $n_i \leftarrow$ first node in Q
  $P(k) \leftarrow \{n_i\}$
  remove n_i from Q /* Add to $P(k)$ all the nodes that are not *connected* with n_i */
  foreach node $n_j \in Q$ **and** $Q \neq \emptyset$ **do**
    if (there is no path $n_i \rightarrow n_j$) **and** (there is no path $n_j \rightarrow n_i$) **then**
      $P(k) \leftarrow P(k) \bigcup \{n_j\}$
      remove n_j from Q
    end if
  end for
  $k \leftarrow k + 1$
end while
return $receivers \leftarrow P(1), P(2), \ldots, P(k-1)$

convenient. We will base our cost model on the effects of lock contention. Suppose that there is high contention for a particular lock. All the statements moved to premutex nodes will not be affected by it because they execute before acquisition of the lock. However, statements moved to the postmutex node will be delayed if there is contention because they execute after the lock has been released. Therefore, when a statement can be moved to both the premutex and postmutex nodes, the premutex node is selected.

The basic mechanism for moving statements outside mutex bodies can be used to move lock-independent control structures. Control structures are handled by checking and aggregating all the nodes contained in the structure into a single super-node and treating it like a single statement. After this process, Algorithm 2 can be used to hoist the structures outside mutex bodies [12].

4.2 LICM for Expressions

If hoisting statements or control structures outside mutex bodies is not possible, it may still be possible to consider moving lock-independent sub-expressions outside mutex bodies. This strategy is similar to moving statements (Algorithm 2) with the following differences:

1. Sub-expressions do not define variables. They only read variables or program constants.
2. If a sub-expression is moved from its original location, the computation performed by the expression must be stored in a temporary variable created by the compiler. The original expression is then replaced by the temporary variable. This is the same substitution performed by common sub-expression and partial redundancy elimination algorithms [1, 3].
3. Contrary to the case with statements and control structures, expressions can only be moved against the flow of control. The reason is that the value computed by the expression needs to be available at the statement containing the original expression.

Algorithm 2 Lock-Independent Code Motion for Statements (LICMS).

INPUT: A CCFG $G = \langle N, E, Entry_G, Exit_G \rangle$ in CSSAME form with pre and postmutex nodes inserted in every mutex body

OUTPUT: The program with lock-independent statements moved to the corresponding premutex and postmutex nodes

foreach lock variable L_i **do**
  foreach mutex body $B_{L_i}(N) \in MutexStruct(L_i)$ **do**
    $n_i \leftarrow node(L_i)$
    foreach lock-independent statement s reached by n_i **do**
      $D_s \leftarrow$ variables defined by s
      $U_s \leftarrow$ variables used by s
      /* Determine which premutex nodes can receive s. */
      $P \leftarrow$ receivers of s at premutex nodes (Algorithm 1)
      foreach $P_i \in P$ **do**
        foreach node $n \in P_i$ **do**
          if (any path between n and $node(s)$ defines or uses a variable in D_s) **or** (any path between n and $node(s)$ defines a variable in U_s) **then**
            remove P_i from P
          end if
        end for
      end for
      /* Determine which postmutex nodes can receive s. */
      $X \leftarrow$ receivers of s at postmutex nodes
      foreach $X_i \in X$ **do**
        foreach node $x \in X_i$ **do**
          if (any path between x and $node(s)$ defines or uses a variable in D_s) **or** (any path between x and $node(s)$ defines a variable in U_s) **then**
            remove X_i from X
          end if
        end for
      end for
      /* Sets P and X contain sets of premutex and postmutex nodes that can receive s. */
      if $P \neq \emptyset$ **then**
        select one $P_i \in P$ (cost model or random)
        remove s from its original location
        replicate s to each node $n \in P_i$
      else if $X \neq \emptyset$ **then**
        select one $X_i \in X$ (cost model or random)
        remove s from its original location
        replicate s to each node $x \in X_i$
      end if
    end for
    /* Remove the mutex body if it is empty. */
    if $B_{L_i}(N) = \emptyset$ **then**
      remove all the `lock` and `unlock` nodes of $B_{L_i}(N)$
    end if
  end for
end for

Algorithm 3 finds and removes lock-independent expressions from mutex bodies in the program. The process of gathering candidate expressions is similar to that of SSAPRE, an SSA based partial redundancy elimination algorithm [3]. Mutex bodies are scanned for lock-independent first-order expressions, which are expressions that contain only one operator. Higher order expressions are handled by successive iterations of the algorithm.

Algorithm 3 Lock-Independent Code Motion for Expressions (LICME).

INPUT: A CCFG in CSSAME form
OUTPUT: The graph with lock-independent expressions moved to the corresponding premutex nodes

1: **repeat**
2:   **foreach** lock variable L_i **do**
3:     **foreach** mutex body $B_{L_i}(N) \in M_{L_i}$ **do**
4:       $E \leftarrow E \bigcup$ set of lock-independent expressions in $B_{L_i}(N)$.
5:       **if** $E \neq \emptyset$ **then**
6:         **foreach** expression $E_j \in E$ **do**
7:           $P \leftarrow$ premutex receivers for E_j (Algorithm 1)
8:           $candidates \leftarrow \emptyset$
9:           **foreach** $P_i \in P$ **do**
10:             **if** $\forall n \in P_i : (n\ DOM\ node(E_j))$ **or** $(node(E_j)\ PDOM\ n)$ **then**
11:               $candidates \leftarrow P_i$
12:               stop looking for candidates
13:             **end if**
14:           **end for**
15:           **if** $candidates \neq \emptyset$ **then**
16:             insert the statement $t_j = E_j$ in all the premutex nodes for `lock` nodes in $candidates$
17:           **end if**
18:         **end for**
19:       **end if**
20:     **end for**
21:   **end for**
22:   /* Replace hoisted expressions inside each mutex body. */
23:   **foreach** lock variable L_i **do**
24:     **foreach** mutex body $B_{L_i}(N) \in M_{L_i}$ **do**
25:       replace hoisted expressions in $B_{L_i}(N)$ with their corresponding temporaries
26:     **end for**
27:   **end for**
28: **until** no more changes have been made

Once lock-independent expressions are identified, the algorithm looks for suitable premutex or postmutex nodes to receive each expression. We observe that since expressions can only be hoisted up in the graph, it is not necessary to consider postmutex nodes when moving lock-independent expressions. Only `lock` nodes are considered by the algorithm. Furthermore, the candidate `lock` must dominate or be post-dominated by the node holding the expression (lines $7 - 13$ in Algorithm 3).

The acceptable receiver sets are stored in the set $candidates$. It can be shown that in this case, the algorithm for computing receiver premutex nodes (Algorithm 1) will find none or exactly one set of `lock` nodes that can receive the expression in their premutex nodes [12].

Figure 1 shows an example program before and after running the LICM algorithm. When LICM is applied to the program in Figure 1(a), the first phase of the algorithm moves the statement at line 6 and the assignment $j = 0$ to the premutex node. The statement at line 10 is sunk to the postmutex node resulting in the equivalent program in Figure 1(b). There is still some lock-independent code in the mutex body, namely the expressions $j < M$ at line 7, the statement j++ at line 7 and the expression $y[j] + sqrt(a) * sqrt(b)$ at line 8. The only hoistable expression is $sqrt(a) * sqrt(b)$ because it is the only expression with all its reaching definitions outside the mutex body (Figure 1(c)). Note that a loop-invariance transformation would have detected this expression and hoisted it out of the loop. LICM goes a step further and hoists the expression outside the mutex body.

```
1 double X[]; /* shared */
2 parloop (i, 0, N) {
3   double a, b; /* local */
4   double y[]; /* local */
5   lock(L);
6   b = a * sin(a);
7   for (j = 0; j < M; j++) {
8     X[j] = y[j] + sqrt(a)
                  * sqrt(b);
9   }
10  a = y[j];
11  unlock(L);
12 }
```

(a) Program before LICM.

```
1 double X[]; /* shared */
2
3 parloop (i, 0, N) {
4   double a, b; /* local */
5   double y[]; /* local */
6
7   ...
8   b = a * sin(a);
9   j = 0;
10  lock(L);
11  for (; j < M; j++) {
12    X[j] = y[j] + sqrt(a)
                  * sqrt(b);
13  }
14  unlock(L);
15  a = y[i];
16  ...
17 }
```

(b) LICM on statements.

```
1 double X[]; /* shared */
2 parloop (i, 0, N) {
3   double a, b; /* local */
4   double y[]; /* local */
5   b = a * sin(a);
6   j = 0;
7   t1 = sqrt(a) * sqrt(b);
8   lock(L);
9   for (; j < M; j++) {
10    X[j] = y[j] + t1;
11  }
12  unlock(L);
13  a = y[j];
14 }
```

(c) LICM on expressions.

Fig. 1. Effects of lock-independent code motion (LICM).

The individual LICM algorithms can be combined into a single LICM algorithm. There are four main phases to the algorithm. The first phase looks for mutex bodies that have nothing but lock-independent nodes. These are the simplest cases. If all the nodes in a mutex body are lock-independent, then the `lock` operations at the lock nodes and the `unlock` operations in the body can be removed. The next three phases move interior lock-independent statements, control structures and expressions outside the mutex bodies in the program.

5 Mutex Body Localization

Consider a mutex body B_L that modifies a shared variable V (Figure 2(a)). With the exception of the definition reaching the unlock node of B_L, all the modifications done to V inside the mutex body can only be observed by the thread. Therefore, it is legal to create a local copy of V and replace all the references to V inside the mutex body to references to the local copy. We call this transformation *mutex body localization* (MBL).

```
double V = 0;
parloop (i, 0, N) {
  double x, y[];
  int i;

  ...
  lock(L);
  i = 0;
  while (V <= x) {
    V = V + y[i++];
  }
  unlock(L);
  ...
}
```

(a) A mutex body before localization.

```
double V = 0;
parloop (i, 0, N) {
  double x, y[], p_V;
  int i;
  ...
  lock(L);
  p_V = V;
  i = 0;
  while (p_V <= x) {
    p_V = p_V + y[i++];
  }
  V = p_V;
  unlock(L);
  ...
}
```

(b) After localization.

```
double V = 0;
parloop (i, 0, N) {
  double x, y[], p_V;
  int i;
  ...
  lock(L);
  p_V = 0;
  i = 0;
  while (p_V <= x) {
    p_V = p_V + y[i++];
  }
  V = V + p_V;
  unlock(L);
  ...
}
```

(c) After reduction recognition.

```
double V = 0;
parloop (i, 0, N) {
  double x, y[], p_V;
  int i;
  ...
  p_V = 0;
  i = 0;
  while (p_V <= x) {
    p_V = p_V + y[i++];
  }
  lock(L);
  V = V + p_V;
  unlock(L);
  ...
}
```

(d) After LICM.

Fig. 2. Applications of mutex body localization.

While LICM looks for lock-independent code, MBL creates lock-independent code by introducing local copies of a shared variable. The basic transformation is straightforward:

1. At the start of the mutex body a local copy of the shared variable is created if there is at least one use for the variable with reaching definitions outside the mutex body.
2. At the mutex body exits, the shared copy is updated from the local copy of the variable if at least one internal definition of the variable reaches that particular unlock node.
3. All the interior references to the shared variable are modified so that they reference the local copy.

Notice that this transformation is legal provided that the affected references are always made inside mutex bodies. Otherwise, the transformation might prevent memory interleavings that were allowed in the original program.

Algorithm 5 makes local copies of a variable a inside a mutex body $B_L(N)$ if the variable can be localized. To determine whether the variable a can be localized it calls Algorithm 4 (a subroutine of Algorithm 5) which returns TRUE if a can be localized inside mutex body $B_L(N)$. The localization algorithm relies on two data structures that can be built during the π rewriting phase of the CSSAME algorithm:

$exposedUses(N)$ is the set of upward-exposed uses from the mutex body $B_L(N)$. This set is associated with the entry nodes in N.

$reachingDefs(X)$ is the set of definitions that can reach the exit nodes X of $B_L(N)$.

Algorithm 5 starts by checking whether the variable can be localized (lines $1-4$). It then checks where the local copies are needed. If there are upward-exposed uses of a, a copy is needed at the start of the mutex body (lines $5-16$). If there are definitions of

a reaching an exit node, the shared copy of a must be updated before exiting the mutex body (lines $17 - 29$). The final phase of the algorithm updates the interior references to a to be references to p_a (lines $30 - 34$). After this phase, the CSSAME form for the program has been altered and it should be updated. The simplest way to do this is to run the CSSAME algorithm again. However, this might be expensive if the localization process is repeated many times.

An alternate solution is to incrementally update the CSSAME form after the variable has been localized. Although this is generally considered a hard problem, the following are some guidelines that should be considered when performing an incremental update of the CSSAME form:

1. If the local copy is created at the start of the mutex body, the statement $p\_a = a$ contains a use of a. This use of a will have the same control reaching definition that the upward-exposed uses of a have. Notice that all the upward-exposed uses of a have the same control reaching definition.
   Since this statement has a conflicting use of a, it requires a π function. The argument list to this π function is the union of all the arguments to all the π functions for a inside the mutex body. Notice that the π functions for a should be for upward-exposed uses of a. This is because the program is in CSSAME form and all conflicting references to a are made inside mutex bodies of the same mutex structure (i.e., a is localizable).
2. All the π functions for a inside the mutex body must disappear because all the interior references to a are replaced by references to p_a.
3. All the interior ϕ functions for a must be converted into ϕ functions for p_a.
4. If the shared copy is updated at the end of the mutex body, the statement $a = p\_a$ contains a use of p_a whose control reaching definition should be the definition of p_a reaching the exit node x.

Algorithm 4 Localization test ($localizable$).

INPUT: A variable a and mutex body $B_L(N)$
OUTPUT: TRUE if a can be localized in $B_L(N)$, FALSE otherwise

$M_L \leftarrow$ mutex structure containing $B_L(N)$
/* Check every conflicting reference r to a in the program. All the conflicting */
/* references to a must occur inside mutex bodies of M_L, otherwise a is not localizable. */
foreach conflicting reference $r \in Refs(a)$ **do**
  /* If we cannot find r in any of the mutex bodies of M_L, then a is not localizable. */
  $protected \leftarrow$ FALSE
  foreach mutex body $B'_L(N') \in M_L$ **do**
    if $node(r)$ is reached by some `lock` node in N' **then**
      $protected \leftarrow$ TRUE
    end if
  end for
  if not $protected$ **then**
    return FALSE
  end if
end for
/* All the references to a are protected. Therefore, a is localizable. */
return TRUE

Algorithm 5 Mutex body localization.

INPUT: (1) An explicitly parallel program P in CSSAME form, (2) A variable a to be localized, (3) A mutex body $B_L(N)$
OUTPUT: $B_L(N)$ with variable a localized

/* Check if a can be localized (Algorithm 4) */
if not $localizable(a, B_L(N))$ **then**
  return
end if
/* Check for upward-exposed uses of a. Since the program is in CSSAME form, */
/* upward-exposed uses have already been computed. If there are */
/* upward-exposed uses of a then we need to make a local copy of a at the start of $B_L(N)$. */
$needEntryCopy \leftarrow$ FALSE
foreach use $u \in exposedUses(N)$ **do**
  if u is a use of a **then**
    $needEntryCopy \leftarrow$ TRUE
  end if
end for
if $needEntryCopy$ **then**
  insert the statement $p\_a = a$ at the start of the mutex body
end if
/* Check if any definition of a reaches the exit nodes of $B_L(N)$. */
/* Since the program is in CSSAME form, the definitions that reach the exit nodes X */
/* have already been computed. If a definition */
/* of a reaches x, we need to make a copy of a before leaving the mutex body. */
$needExitCopy \leftarrow$ FALSE
foreach definition $d \in reachingDefs(X)$ **do**
  if d is a definition of a **then**
    $needExitCopy \leftarrow$ TRUE
  end if
end for
if $needExitCopy$ **then**
  insert the statement $a = p\_a$ at the exit nodes of the mutex body
end if
/* Update references to a inside the mutex body to reference */
/* the local version p_a instead of the shared version a. */
foreach reference to a inside $B_L(N)$ **do**
  replace a with p_a
end for
update CSSAME information for all references to p_a inside $B_L(N)$

The MBL transformation by itself does not necessarily improve the performance of a program but it opens up new optimization opportunities. The main benefit of localization is that it might create more lock-independent code. For instance, if a thread contains read-only references to a variable V, localizing V will make those reads into lock-independent operations which in turn might make the whole statement lock-independent. Consider the sample program in Figure 2(a). After localization (Figure 2(b)), most statements inside the mutex body for L are lock-independent. However, none can be moved outside because of the read and write operations to the shared variable V at the fringes of the mutex body. If the compiler incorporates a reduction recognition pass, it is possible to do the reduction locally and only update V at the end (Figure 2(c)). Now all the lock-independent code in the mutex body can be moved to the premutex node resulting in the equivalent program in Figure 2(d).

6 Experimental results

The algorithms discussed in this paper have been implemented[2] in a prototype compiler for the C language using the SUIF compiler system [5]. Our runtime system leverages on the SUIF runtime system to execute the parallel program.

Once the program has been parsed by the SUIF front-end, the compiler creates the corresponding CCFG and its CSSAME form. We do not transform the input program into SSA form. Instead we use factored use-def chains [17] in the flowgraph and display the source code annotated with the appropriate π and ϕ functions (variables are not renamed but referenced using line number information in the corresponding π or ϕ functions). The CCFG can be displayed using a variety of graph visualization systems. The CSSAME form for the program can also be displayed as an option. Finally, the compiler incorporates mutual exclusion validation techniques to warn the user about potential problems with the synchronization structure of the program [14].

Synchronization overhead is sometimes exacerbated by an expensive implementation of `lock` and `unlock` operations. To address this problem, several techniques have been proposed to implement more efficient locking primitives [2, 16]. But there is another source of overhead that even the most efficient implementation cannot alleviate: contention. Lock contention occurs when the demand for a particular lock variable is so high that threads spend a significant amount of time waiting for other threads to release the lock. The techniques for eliminating superfluous synchronization operations developed in this paper can complement the benefits of using an efficient locking mechanism.

6.1 Water

The Water application simulates forces and potentials in a system of liquid water molecules. The simulation is done over a specified number of time-steps until the system reaches equilibrium. Mutual exclusion synchronization is used when computing inter-molecular interactions and for keeping a global sum that is computed every time-step.

To study the effects of LICM in Water, we performed experiments that varied the total number of molecules (N), the number of molecule locks (ML), and, the number of simulation time-steps (TS). Experiments were performed on an SGI PowerChallenge with 8 processors and 384Mb of memory. The implementation uses SGI native threads (`sproc`) and hardware locks (`ulock`). All the experiments were executed on 8 processors with no other system activity.

The first experiment studies the performance effects of LICM as a function of synchronization overhead. As the number of time-steps increases, so does synchronization overhead. Table 1 shows the speedups obtained as a function of the number of time-steps and number of molecules simulated. Notice how the speedups obtained with LICM are lower when a larger number of molecules are simulated. This is caused by the larger computation to synchronization ratio in the larger problem. Also, by restricting the number of molecule locks available we are increasing lock contention. Naturally, as the number of available locks increases, the effects of LICM are diminished.

[2] A preliminary version is available at `http://www.cs.ualberta.ca/~jonathan/CSSAME/`

	64 molecules (10 molecule locks)			216 molecules (10 molecule locks)		
Time steps	no LICM time (secs)	with LICM time (secs)	Relative Speedup	no LICM time (secs)	LICM time (secs)	Relative Speedup
70	157	144	1.09	1527	1463	1.04
80	183	171	1.07	1772	1763	1.00
100	235	219	1.07	2344	2285	1.02
120	296	269	1.10	2827	2809	1.00

Table 1. Speedups obtained by LICM on Water as a function of the number of simulation time-steps.

Since molecule locks are accessed more as the number of time-steps increases, the contention on these locks also increases. To measure lock contention we used the hardware timers provided by the system to measure the average delay of acquiring a lock. We then computed the average delay over the 10 molecule locks. This is shown in Table 2. This table shows how average lock contention on the molecule locks increases as a function of the number of simulation time-steps. Notice that although LICM reduces lock contention significantly, its impact on the runtime of the program may not be too noticeable if the ratio of computation to synchronization is high enough. Again notice how lock contention decreases with the larger problem size. This explains the diminished effects of LICM on large problems.

	64 molecules			216 molecules		
Time steps	no LICM avg delay (μsecs)	with LICM avg delay (μsecs)	Ratio	no LICM avg delay (μsecs)	with LICM avg delay (μsecs)	Ratio
70	699	72	9.71	561	68	8.25
80	712	73	9.75	575	72	7.99
100	718	71	10.11	557	70	7.96
120	729	85	8.58	564	62	9.10

Table 2. Effects of LICM on lock contention in Water.

6.2 Ocean

Ocean studies eddy and boundary currents in large-scale ocean movements. Mutual exclusion is used to update global sums and to access a global convergence flag used in the iterative solver. The update of global sums is done with the same strategy used in Water. A local sum is computed and aggregated to the global sum.

To study the effect of MBL and LICM on this application, we simplified some routines in Ocean to compute global sums directly (the original program computes global sums by aggregating locally computed partial sums). We named this new version Simple Ocean. The intention is to demonstrate how some of the optimizations that are traditionally performed manually by the programmer can be automated using the techniques developed in this paper. Table 3 shows the performance improvements obtained by applying MBL and LICM to Simple Ocean. The program was executed on 8 processors with four different ocean sizes and a time-step of 180 seconds.

Ocean size	no MBL+LICM time (sec)	with MBL+LICM time (sec)	Relative Speedup
66×66	21	19	1.11
130×130	69	56	1.23
258×258	258	198	1.30
514×514	865	787	1.10

Table 3. Effects of MBL and LICM on Simple Ocean.

The performance improvements obtained on Simple Ocean using MBL and LICM are the same improvements obtained by the manual optimizations done in the original program. The important point of this experiment is to show that using the techniques developed in this paper it is possible to automatically optimize inefficient synchronization patterns. We do not expect experienced programmers to write such inefficient synchronization, but this kind of code could be found in programs written by a less experienced programmer or generated from generic code templates in a programming environment.

7 Conclusions and future work

We have shown how the CSSAME form allows new optimization opportunities by taking advantage of the semantics imposed by mutual exclusion synchronization. In previous work we have shown how the reduction of memory conflicts across threads can improve the effectiveness of adapted scalar optimization strategies [13]. In this paper, we have introduced two new optimization techniques that are specifically targeted at explicitly parallel programs: *Lock-Independent Code Motion* (LICM) moves code that does not need to be locked outside critical sections and *Mutex Body Localization* (MBL) converts shared memory references into local memory references. We consider these techniques a step towards a unified analysis and optimization framework for explicitly parallel programs. In turn this should facilitate the adoption of high-level systems with language-supported parallelism and synchronization. These systems typically provide powerful abstractions that make parallel programming easier, but those same abstractions often hinder performance. Experienced programmers recognize these limitations and manually circumvent them by removing abstraction layers to speed-up their code. With the techniques developed in this paper, we can transfer the burden of these transformations to the compiler.

References

[1] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Reading, Mass.: Addison-Wesley, Reading, MA, second edition, 1986.

[2] D. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin Locks: Featherweight Synchronization for Java. In *ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 258–268, Montreal, Canada, June 1998.

[3] F. Chow, S. Chan, R. Kennedy, S.-M. Liu, R. Lo, and P. Tu. A new algorithm for partial redundancy elimination based on SSA form. In *ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, Las Vegas, 1997.

[4] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[5] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S. Liao, E. Bugnion, and M. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, 29(12):84–89, December 1996.

[6] J. Knoop and B. Steffen. Code motion for explicitly parallel programs. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, GA, May 1999.

[7] J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Transactions on Programming Languages and Systems*, 18(3):268–299, May 1996.

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

[9] J. Lee, S. Midkiff, and D. A. Padua. Concurrent static single assignment form and constant propagation for explicitly parallel programs. In *Proceedings of the Tenth Workshop on Languages and Compilers for Parallel Computing*, August 1997.

[10] J. Lee, D. A. Padua, and S. Midkiff. Basic compiler algorithms for parallel programs. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, GA, May 1999.

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

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

[13] D. Novillo, R. Unrau, and J. Schaeffer. Concurrent SSA Form in the Presence of Mutual Exclusion. In *1998 International Conference on Parallel Processing*, pages 356–364, Minneapolis, Minnesota, August 1998.

[14] D. Novillo, R. Unrau, and J. Schaeffer. Identifying and Validating Irregular Mutual Exclusion in Concurrent Programs. In *European Conference on Parallel Computing (Euro-Par 2000)*, August 2000.

[15] J. Singh, W. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[16] R. Unrau, O. Krieger, B. Gamsa, and M. Stumm. Experiences with Locking in a NUMA Multiprocessor Operating System Kernel. In *Proceedings for the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI '94)*, pages 139–152, 1994.

[17] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Reading, Mass.: Addison-Wesley, Redwood City, CA, 1996.