University of Alberta

Analysis and Optimization of Explicitly Parallel Programs

by

Diego Novillo Ronald C. Unrau Jonathan Schaeffer

Technical Report TR 98-11
August 1998

DEPARTMENT OF COMPUTING SCIENCE
University of Alberta
Edmonton, Alberta, Canada

Contents

List of Algorithms

Analysis and Optimization of Explicitly Parallel Programs

Diego Novillo[†] Ronald C. Unrau[‡] Jonathan Schaeffer[†]

[†]{diego,jonathan}@cs.ualberta.ca
Department of Computing Science
University of Alberta
Canada

[‡]runrau@cygnus.com
Cygnus Solutions Ltd.
Sunnyvale, CA
U.S.A.

A shorter version of this document has been submitted to the Journal of Parallel and Distributed Computing Special Issue on Compilation and Architectural Support for Parallel Applications (1999)

Abstract

Most current compiler analysis techniques are unable to cope with the semantics introduced by explicit parallel and synchronization constructs in parallel programs. In this paper we introduce new analysis and optimization techniques for compiling explicitly parallel programs that use mutual exclusion synchronization. We introduce the CSSAME form, an extension of Concurrent Static Single Assignment (CSSA) that incorporates mutual exclusion into a data flow framework for explicitly parallel programs. We show how this analysis can improve the effectiveness of constant propagation in a parallel program. We also present a modification to a sequential dead code elimination algorithm to work on explicitly parallel programs.

Finally, we introduce new optimization techniques specifically targeted at explicitly parallel programs. These techniques apply optimizing transformations to a program by taking advantage of its parallel and synchronization structure. We prove the correctness of these transformations and provide algorithms that implement them. The techniques presented in this paper have been implemented by extending the SUIF compiler system.

1 Introduction

Although recent advances in parallelizing compilers and data-parallel languages have been impressive, there are important problem domains for which parallelizing the best sequential algorithm yields sub-optimal performance relative to an implementation that is explicitly parallel from the outset. Furthermore, popular high-level programming languages like Java incorporate parallel constructs at the language level. For these reasons, we believe that there is a need for compilers that optimize these programs, and that the demand for such compilers will increase.

There are two important issues that must be addressed by compilers that optimize explicitly parallel programs. First, these compilers should take advantage of the vast number of optimizations developed for sequential languages. Unfortunately, this is not a trivial task. Sequential optimization techniques cannot be directly applied to explicitly parallel programs because they may generate incorrect transformations [18]. Therefore, these techniques must be adapted to work on explicitly parallel programs. Second, these compilers must have an innate knowledge of the parallel constructs, synchronization operations and shared memory semantics present in explicitly parallel programs. Understanding how these three elements interact not only allows a safe translation of sequential optimizations but also provides opportunities for new optimizations of explicitly parallel programs.

The main goal of our work is to develop a unified analysis and optimization framework that allows the translation of sequential optimization techniques to explicitly parallel code and the development of new optimizations that take advantage of the parallel structure of these programs. In this paper we give a first step towards that goal. We present an analysis framework for explicitly parallel programs with mutual exclusion synchronization and interleaving memory semantics based on the Static Single Assignment form. We adapt sequential optimization techniques to work on explicitly parallel programs and develop new optimization techniques that take advantage of the parallel and synchronization structure of these programs. Specifically, we

1. extend the concurrent control flow graph used by Lee *et al.* [15] (Section 4.1) and show how to detect mutual exclusion synchronization in a parallel program (Section 4.3),

2. introduce the CSSAME[1] form, an extension to the CSSA form [15] to account for the semantics introduced by mutual exclusion synchronization (Section 6),

3. show how CSSAME can improve the effectiveness of the Concurrent Sparse Conditional Constant (CSCC) propagation algorithm [15] (Section 7.1),

4. adapt a sequential dead code elimination algorithm to work on explicitly parallel programs (Section 7.2),

5. introduce new optimization techniques for explicitly parallel programs: lock independent code motion (Section 7.3), mutex body localization (Section 7.4), single-writer multiple-readers code motion (Section 7.4.1), code sinking (Section 7.5), lock picking (Section 7.6) and lock partitioning (Section 7.7).

[1]Pronounced *sesame*.

2 Related work

Recent developments in this area have started to uncover the potential benefits of analysis and optimization techniques for explicitly parallel programs. Shasha and Snir proposed an analysis technique called *cycle detection* that allows re-ordering of memory references in a program to increase concurrency while maintaining the sequential consistency dictated by the code [20]. Krishnamurthy and Yelick extended cycle detection analysis to incorporate additional information from synchronization in the program [13]. Although their work supports post-wait, barrier and mutual exclusion synchronization, they only focus on optimizing remote memory references on a specific class of explicitly parallel programs.

Grunwald and Srinivasan developed data-flow equations to compute reaching definition information on explicitly parallel programs with `cobegin/coend` parallel sections [7]. However, their work can only tolerate the weak memory consistency model dictated by the PCF Fortran standard. Parallel sections are required to be data independent; memory updates are done at specific points in the program using the `copy-in/copy-out` model. Their work only deals with event-based synchronization using `set` and `wait` operations. The same memory and synchronization model is used by the Static Single Assignment (SSA) framework developed by Srinivasan, Hook and Wolfe for explicitly parallel programs [22].

Stronger memory models have also been considered. Knoop, Steffen and Vollmer developed a bitvector analysis framework for parallel programs with shared memory and interleaving semantics [12]. They show how to adapt standard optimization algorithms to their framework. However, they do not incorporate synchronization operations in their analysis. Lee, Midkiff and Padua propose a Concurrent SSA framework (CSSA) for explicitly parallel programs and interleaving memory semantics [15]. Their work only considers event-based synchronization operations and imposes some restrictions on the input program.

Despite the growing interest in this area, existing techniques are still in their primitive stages, especially when compared to their sequential counterparts. A major limitation of most existing analysis and optimization techniques is the restricted knowledge about synchronization in the program. With the exception of Krishnamurthy and Yelick's work, existing analysis frameworks only recognize a subset of event-based synchronization (i.e., `set` and `wait`, usually with no `clear`). We see this as a severe limitation because event synchronization can only be used to describe a small class of parallel algorithms. Our work extends Lee *et al.*'s to incorporate mutual exclusion analysis for explicitly parallel programs.

3 Language model

There are a variety of models of parallelism, memory semantics and synchronization methods. In this section we define the model we use and the assumptions we make about the underlying execution environment.

An explicitly parallel program starts as a single thread of computation. New threads are logically created when execution reaches a parallel section. Although the creation, placement and scheduling of threads is not significant for our research, the compiler must be able to recognize parallel sections in the code. In this paper we assume the following:

1. *Parallelism*. The focus of this paper is task-parallel programs. Parallel sections are specified using the `cobegin`/`coend` construct (Figure 1).

2. *Memory model*. Threads run in a shared address space with interleaving semantics (i.e., updates to shared memory made by one thread are immediately visible to other threads). Programs share memory via shared variables. Pointers, arrays and aliasing issues will not be considered in this paper.

3. *Synchronization*. Both event-based and mutual exclusion synchronization are supported. Mutual exclusion is used to serialize references to shared variables in the program. We will assume, without loss of generality, that programmers use standard `lock` and `unlock` instructions to serialize access to shared variables. Event synchronization is supported using `set` and `wait` instructions. All the support for event synchronization is derived from the algorithms in [14].

4 Mutual exclusion analysis

In an explicitly parallel program with interleaving memory semantics, the use of a shared variable v can be reached by any definition of v in another concurrent thread. However, mutual exclusion may prevent some variable definitions from being visible in other threads. For example, consider the program in Figure 1[2]. If we ignore the mutual exclusion regions created by the locks we will conclude that the definition for variable a in thread T_0 can reach both uses of a in thread T_1. However, the synchronization used in the program serializes the references to a so that the assignment to a in T_0 cannot reach the second use of a in T_1. Therefore, the call to function $g()$ in T_1 will always be executed with $a = 3$.

[2]Unless otherwise stated, the example programs presented in this paper are to be considered complete programs.

```
cobegin /* Begin concurrent execution */
  T_0: begin     /* Launch thread T_0 */
    if (b > 0) {
      b = 3 / a;
    }
    lock(L);
    a = a + b;
    unlock(L);
  end

  T_1: begin        /* Launch thread T_1 */
    f(a);
    lock(L);
    a = 3;            /* This kills the assignment to a in T_0 */
    b = b + g(a); /* Variable a is always 3 */
    unlock(L);
  end
coend
```

Figure 1: Mutual exclusion can reduce data dependencies across threads in a parallel program.

The following sections describe the data structures and algorithms used to identify mutual exclusion synchronization in an explicitly parallel program. Section 4.1 describes concurrent control flow graphs and Section 4.3 describes the algorithm used to identify mutual exclusion synchronization.

4.1 Concurrent Control Flow Graph

We extend the Concurrent Control Flow Graph (CCFG) [15] to represent mutual exclusion synchronization. In particular, we incorporate undirected mutex synchronization edges which represent mutual exclusion constraints but do not enforce a specific execution order. We also extend the concept of concurrent basic block so that each `lock` and `unlock` operation is represented by a separate node in the CCFG. Mutex synchronization edges join `lock` and `unlock` nodes that operate on the same variable in concurrent threads. An example of a CCFG is shown in Figure 2 (some conflict edges have been removed to improve readability).

Definition 4.1 (shared variable conflicts) Two variable references in different threads *conflict* if both reference the same variable, one of them is a write reference and the threads can execute concurrently. □

Definition 4.2 (concurrent basic block) A *concurrent basic block* is a basic block [1] with the following additional properties

1. Only the first statement of the block can be a `wait` statement or contain a use of a conflicting variable.

```
a = 0;
b = 0;
cobegin
  T_0: begin
    lock(L);
    a = 5;
    b = a + 3;
    if (b > 4) {
      a = a + b;
    }
    x = a;
    unlock(L);
  end

  T_1: begin
    lock(L);
    a = b + 6;
    y = a;
    unlock(L);
  end
coend
print(x, y);
```

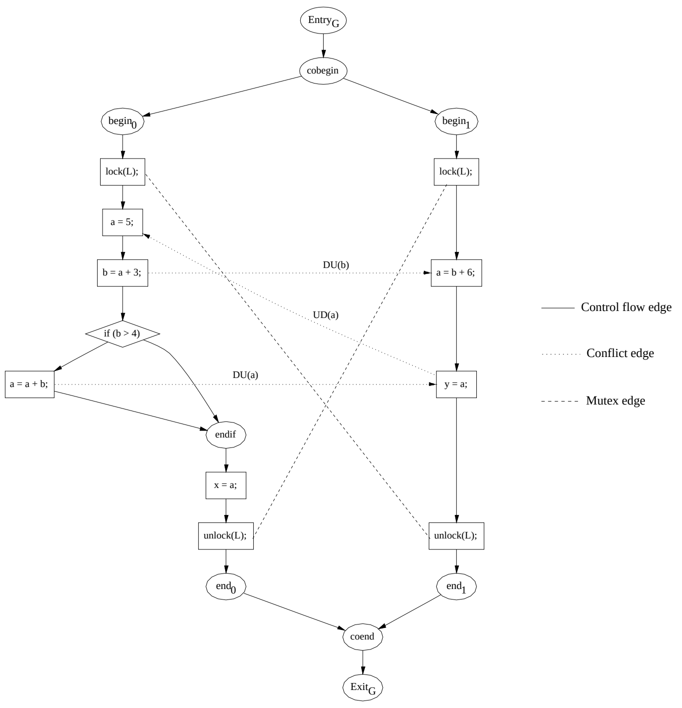


Figure 2: A parallel program and its Concurrent Control Flow Graph.

2. Only the last statement of the block can be a `set` statement or contain a definition of a conflicting variable.

3. The following synchronization operations will be the only statement in the block: `lock` and `unlock`.

4. The following instructions will be the only statement in the block: `cobegin` and `coend`. □

Definition 4.3 (conflicts between concurrent basic blocks) Two concurrent basic blocks a and b in different threads *conflict* if they can execute concurrently and contain conflicting variable references. □

Definition 4.4 (Concurrent Control Flow Graph) A *Concurrent Control Flow Graph (CCFG)* is a directed graph $G = \langle N, E, Entry_G, Exit_G \rangle$ such that:

1. N is the set of nodes in the graph. Each node in N corresponds to a concurrent basic block.

2. $Entry_G$ and $Exit_G$ are the unique entry and exit points of the program (nodes labeled Entry and Exit in Figure 2).

3. $E = E_{ct} \bigcup E_{sy} \bigcup E_{cf}$ is the set of edges in the graph such that:

   (a) E_{ct} is the set of control flow edges. These edges have the same meaning as in a sequential Control Flow Graph (solid lines in Figure 2).

   (b) $E_{sy} = E_{mutex} \bigcup E_{dsync}$ is the set of synchronization edges. Two different kinds of synchronization are recognized:

      i. E_{mutex} is the set of mutex synchronization edges representing mutual exclusion constraints. Mutex synchronization edges are undirected edges between related `lock` and `unlock` operations (dashed lines in Figure 2).

      ii. E_{dsync} is the set of directed synchronization edges representing ordering constraints. These edges join related `set` and `wait` statements in different threads.

   (c) E_{cf} is the set of conflict edges. Conflict edges are directed edges that join any two concurrent basic blocks that conflict. The labels on conflict edges represent the memory operations done at each end of the edge. Each label has the format `TH(V)`. The first letter (`T`) represents the memory operation done at the tail of the edge. The second letter (`H`) represents the operation done at the head of the edge. The third letter (`V`) represents the variable referenced by both operations. Memory operations include definitions (`D`) or uses (`U`) (dotted lines in Figure 2). □

Given a concurrent control flow graph $G = \langle N, E, Entry_G, Exit_G \rangle$ and two nodes $x, y \in G$, we will use the following concepts in subsequent sections:

Definition 4.5 (entry and exit nodes) Given a thread T, $begin_T$ is the entry node for T, end_T is the exit node for T, $cobegin_T$ is the *cobegin* node for the innermost `cobegin`/`coend` structure containing T, and $coend_T$ is the corresponding *coend* node for $cobegin_T$. □

Definition 4.6 (control path) A path from x to y is a *control path* if it only contains edges in E_{ct}. □

Definition 4.7 (dominance) Node x *dominates* node y, denoted $x\ DOM\, y$, if every control path from $Entry_G$ to y contains x. Node x is in the set of dominators of y, denoted $x \in DOM(y)$. Node y is in the set of nodes dominated by x, denoted $y \in DOM^{-1}(x)$. □

Definition 4.8 (strict dominance) Node x *strictly dominates* node y, denoted $x\ SDOM\ y$, if $x\ DOM\, y$ and $x \neq y$. Node x is in the set of strict dominators of y, denoted $x \in SDOM(y)$. Node y is in the set of nodes strictly dominated by x, denoted $y \in SDOM^{-1}(x)$. □

Definition 4.9 (post-dominance) Node y *post-dominates* node x, denoted $y\ PDOM\ x$, if every control path from x to $Exit_G$ contains y. Node y is in the set of post-dominators of x, denoted $y \in PDOM(x)$. Node x is in the set of nodes post-dominated by y, denoted $x \in PDOM^{-1}(y)$. □

Definition 4.10 (strict post-dominance) Node y *strictly post-dominates* node x, denoted $y\ SPDOM\ x$, if $y\ PDOM\ x$ and $x \neq y$. Node y is in the set of strict post-dominators of x, denoted $y \in SPDOM(x)$. Node x is in the set of nodes strictly post-dominated by y, denoted $x \in SPDOM^{-1}(y)$. □

4.2 Building the concurrent control flow graph of a program

The algorithm used to build the concurrent control flow graph for an explicitly parallel program P consists of three phases: placement of control structures, conflict analysis and synchronization analysis. The following sections describe each of these phases in detail.

Algorithm 4.1 Build a concurrent control flow graph.

```
INPUT:   An explicitly parallel program $P$
OUTPUT:  The concurrent control flow graph $G = \langle N, E, Entry_G, Exit_G \rangle$ for $P$

1: Build maximal basic blocks and control edges (Section 4.2.1).
2: Add conflict edges (Algorithm 4.3).
3: Add synchronization edges (Algorithm 4.4).
```

4.2.1 Control structures

Graph nodes and control edges are created using a slightly modified version of a standard algorithm to build control flow graphs. The basic algorithm is an extension to the control flow graph building algorithm provided by the Machine SUIF CFG library [10]. The modification allows the original algorithm to recognize the `cobegin`/`coend` construct. The algorithm is conceptually very simple and it will not be discussed further in this document.

This step builds maximal basic blocks [1], not concurrent basic blocks. Subsequent phases of the algorithm will split the maximal basic blocks to create concurrent basic blocks and incorporate conflict and synchronization edges to the base graph.

4.2.2 Conflict analysis

This phase traverses the graph looking for nodes that can execute concurrently and access the same memory location in a conflicting manner. These nodes are marked conflicting and split up to create concurrent basic blocks. Finally, conflict edges are created to join the conflicting nodes (Algorithm 4.3).

When analyzing two nodes a and b for conflicts or synchronization we must determine whether a and b belong to concurrent threads. That is, barring synchronization and control flow constraints, could a and b execute concurrently? This information is used to avoid placing conflict and synchronization edges across non-concurrent threads. Algorithm 4.2 computes the concurrency relation. The algorithm assumes the existence of two data structures:

1. $Thread(n)$ is the thread that contains node n. Threads are assumed to have a unique id computed automatically by the compiler. The sequential parts of the program are always executed by thread 0.

2. $ParAncestors(n)$ is the set of `cobegin` nodes that can be reached in a backwards traversal of the dominator tree from node n to the entry node of the CCFG.

Algorithm 4.2 Concurrency relation.

INPUT: Two concurrent basic blocks $a, b \in G = \langle N, E, Entry_G, Exit_G \rangle$.
OUTPUT: TRUE if a and b can execute concurrently, FALSE otherwise.

```
1:  function $conc(a, b)$
2:  /* If $a$ or $b$ are in a sequential region, they cannot be concurrent. */
3:  if $Thread(a) = 0 \lor Thread(b) = 0$ then
4:      return FALSE
5:  end if
6:
7:  /* If $a$ and $b$ have a common cobegin node in their */
8:  /* $ParAncestors$ set and they are on different threads */
9:  /* and they are not the same node, they are concurrent. */
10: if $\exists n \in ParAncestors(a)$ s.t. $n =$ cobegin $\land a \neq b \land Thread(a) \neq Thread(b)$ then
11:     return TRUE
12: end if
13:
14: /* None of the previous tests succeeded. The nodes are not concurrent. */
15: return FALSE
```

4.2.3 Synchronization analysis

This phase adds synchronization edges according to Definition 4.4 (Algorithm 4.4). The algorithm uses the same data structure $ParAncestors(n)$ used in algorithm 4.2.

4.3 Mutex structures

The concepts and algorithms described in this section are based on the non-concurrency analysis techniques developed by Masticola and Ryder [16]. Our work differs from theirs in the following aspects:

Algorithm 4.3 Conflict analysis.

INPUT: An incomplete concurrent control flow graph $G = \langle N, E, Entry_G, Exit_G \rangle$ with no conflict edges.
OUTPUT: The CCFG G given as input with conflict edges E_{cf} added.

```
1: $E_{cf} \leftarrow \emptyset$
2: **foreach** $a \in N$ **do**
3:   **foreach** $b \in N$ **do**
4:     **if** $conc(a, b) = $ TRUE $\wedge$ $a$ and $b$ conflict **then**
5:       $E_{cf} \leftarrow E_{cf} \bigcup \{(a, b)\}$
6:     **end if**
7:   **end for**
8: **end for**
9: **foreach** $(a, b) \in E_{cf}$ **do**
10:   Split blocks $a$ and $b$ to comply with definition 4.2.
11: **end for**
```

Algorithm 4.4 Synchronization analysis.

INPUT: An incomplete concurrent control flow graph $G = \langle N, E, Entry_G, Exit_G \rangle$ with no synchronization edges.
OUTPUT: The graph G with synchronization edges $E_{sy} = E_{mutex} \bigcup E_{dsync}$ added.

```
1: $E_{mutex} \leftarrow \emptyset$
2: $E_{dsync} \leftarrow \emptyset$
3:
4: /* Join related locks and events. */
5: **foreach** $a \in N$ **do**
6:   **foreach** $b \in N$ **do**
7:     **if** $conc(a, b) = $ TRUE **then**
8:       **if** $a = lock(l) \wedge b = unlock(l)$ **then**
9:         $E_{mutex} \leftarrow E_{mutex} \bigcup \{(a, b)\}$
10:       **else if** $a = set(s) \wedge b = wait(s)$ **then**
11:         $E_{dsync} \leftarrow E_{dsync} \bigcup \{(a, b)\}$
12:       **end if**
13:     **end if**
14:   **end for**
15: **end for**
16:
17: $E_{sy} \leftarrow E_{mutex} \bigcup E_{dsync}$
```

- Our analysis targets locks instead of binary semaphores.
- Our analysis gathers data flow information for the purposes of program optimization instead of deadlock detection.
- Even though the notation is similar, there are differences in the definitions and the algorithms used. In particular, we use a simpler notion of mutex body that is not based on the concept of *strict interval* defined by Masticola in [17]. Strict intervals require other structural conditions needed for deadlock analysis. For instance, strict intervals do not include ambiguous or illegal mutex bodies. If at the end of the mutex analysis there is at least one unmatched `lock` operation for a lock variable L, the whole set of mutex bodies for L will be discarded. In our case, we allow mutex structures with ill-formed mutex bodies. Our data-flow analysis will still be correct because illegal mutex bodies in a mutex structure will not be considered when reducing data dependencies. Our analysis algorithms only ignore illegal mutex bodies, not the whole mutex structure.
- We do not consider mutex bodies that contain parallel structures and mutex bodies that are not completely contained in a single thread body. Such mutex bodies will be ignored and not affected by our optimizations.

Definition 4.11 (mutex body) Given a CCFG $G = \langle N, E, Entry_G, Exit_G \rangle$, a synchronization variable L and two nodes $n, x \in G$, the set $B_L(n, x) = SDOM^{-1}(n) \bigcap PDOM^{-1}(x)$ is a *mutex body* for L if the following conditions are met:

1. $n =$ `lock(L)` and $x =$ `unlock(L)`,
2. $n \; DOM \, x$ and $x \; PDOM \, n$, and
3. $\forall a \in B_L(n, x)$ such that $a \neq n \wedge a \neq x \Rightarrow a$ cannot be any of the following: `lock(L)`, `unlock(L)`, `cobegin` or `coend`.

A mutex body defines a single-entry, single-exit region of the graph delimited by nodes n and x. The mutex body includes all the nodes strictly dominated by n and post-dominated by x (i.e., node n is not included in $B_L(n, x)$). Notice that this definition may not account for all possible mutual exclusion sections in the code. For instance, the code fragment in Figure 3 will not be classified as a mutex body because the lock operations are not guaranteed to execute (i.e., the region does not have a unique entry and a unique exit point). In this particular case, more detailed analysis might reveal that both conditionals have the same value for every possible execution of the program. However, we currently do not deal with these ambiguously determined mutex bodies, but instead our compiler will issue a warning message when processing the code in Figure 3.

```
if (cond_1) { lock(L); }
statements;
if (cond_2) { unlock(L); }
```

Figure 3: An ambiguous mutex body.

Definition 4.12 (mutex structure) A *mutex structure* for a synchronization object L, denoted M_L, is the set of all mutex bodies $B_L(n, x)$ in the program. □

Definition 4.13 (pure mutex structure) A mutex structure M_L is *pure* if for every `lock(L)` node n there exists an `unlock(L)` node x such that n and x are a mutex body for L. Otherwise, the mutex structure M_L is *impure*. □

Algorithm 4.5 returns the set of all the mutex structures in an explicitly parallel program. There are four main steps in the algorithm:

1. Lines 1–11 traverse the CCFG looking for all the `lock` and `unlock` nodes. Two sets are associated with each lock variable L_i, the set of nodes that lock L_i (p_i^{lock}) and the set of nodes that unlock L_i (p_i^{unlock}). This step complies with condition 1 of Definition 4.11. This step can be computed in $O(m \times me)$ time, where m is the number of lock variables used in the program and me is the number of `lock` and `unlock` nodes in the CCFG.

2. Lines 12–14 call two standard algorithms to build the dominator and post-dominator trees for G [9]. These trees are required to determine dominator information later on. The computation of dominator and post-dominator trees can be done in $O(|E_{ct}|)$ [9].

3. Lines 15–24 traverse the lock and unlock sets for each variable L_i looking for pairs of nodes that comply with condition 2 of Definition 4.11. This finds all the pairs of nodes (n, x) such that $n \; DOM x$ and $x \; PDOM n$. Notice that this step might produce illegal mutex bodies. For instance, consider the following code fragment:

```
(p_1) lock(L);
...
(p_2) unlock(L);
...
(p_3) lock(L);
...
(p_4) unlock(L);
```

The code only defines two distinct mutex bodies (p_1, p_2) and (p_3, p_4). However, this step will identify a third mutex body, (p_1, p_4). The extra mutex body (p_1, p_4) will be removed by the next step. This step can be computed in $O(m \times me^2)$ time.

Algorithm 4.5 Identification of mutex structures.

INPUT: A CCFG $G = \langle N, E, Entry_G, Exit_G \rangle$, a set $L = \{L_1, L_2, \ldots, L_m\}$ containing all the lock variables used in the program and a set N_{me} containing all the nodes that contain a `lock` or an `unlock` instruction.

OUTPUT: A set of mutex structures $M = \{M_1, M_2, \ldots, M_m\}$ where M_i is the set of mutex bodies for lock variable L_i.

```
1: /* Find nodes in $G$ that lock and unlock each $L_i$. */
2: foreach lock variable $L_i$ do
3:   foreach $n \in N_{me}$ do
4:     if $n = \texttt{lock}(L_i)$ then
5:       add $n$ to $p_i^{lock}$
6:     end if
7:     if $n = \texttt{unlock}(L_i)$ then
8:       add $n$ to $p_i^{unlock}$
9:     end if
10:  end for
11: end for
12: /* Build the dominator and post-dominator trees for $G$. */
13: call buildDomTree($G$)
14: call buildPDomTree($G$)
15: /* Find candidate mutex bodies. */
16: foreach lock variable $L_i$ do
17:   foreach $n \in p_i^{lock}$ do
18:     foreach $x \in p_i^{unlock}$ do
19:       if $n \in DOM(x)$ and $x \in PDOM(n)$ then
20:         add $(n, x)$ to the set of candidates $M_i$
21:       end if
22:     end for
23:   end for
24: end for
25: /* Remove illegal mutex bodies from each $M_i$. */
26: foreach $(n, x) \in M_i$ do
27:   foreach node $m$ such that ($n\ SDOM m$) and ($x\ SPDOM m$) do
28:     if ($m = \texttt{lock}(L_i)$) or ($m = \texttt{unlock}(L_i)$) or ($m = \texttt{cobegin}$) or ($m = \texttt{coend}$) then
29:       remove $(n, x)$ from $M_i$
30:     end if
31:   end for
32:   /* Check if $M_i$ is *pure* or *impure*. If all the lock operations for $L_i$ */
33:   /* form a mutex body then $M_i$ is pure. */
34:   if $|p_i^{lock}| = |p_i^{unlock}| = |M_i|$ then
35:     $pure(M_i) \leftarrow$ TRUE
36:   else
37:     $pure(M_i) \leftarrow$ FALSE
38:   end if
39: end for
40: $M \leftarrow \{M_1, M_2, \ldots, M_m\}$
41: return $M$
```

4. Lines 25–32 enforce condition 3 of Definition 4.11. That is, it removes any body found by the previous step that contains `lock` or `unlock` nodes for the same variable (other than the entry and exit nodes for the body). Notice that the mutex body is completely defined by the pair of nodes (n, x). All we have to do to determine if a node $a \in G$ belongs to the mutex body is check whether $n\ SDOM\, a$ and $x\ SPDOM\, a$. This step can be computed in $O(m \times mbsz)$, where $mbsz$ is the average number of nodes in each mutex body of the program.

In general, we expect that the number of lock variables (m), the number of `lock` and `unlock` operations (me) and the size of mutex bodies ($mbsz$) to be significantly smaller than the number of control edges ($|E_{ct}|$) and nodes in the program ($|N|$). Therefore, the execution time of this algorithm should be dominated by the computation of the dominator and post-dominator trees.

Lemma 4.1 (correctness of the mutex structure algorithm) The set M returned by Algorithm 4.5 contains mutex structures for every lock variable in the program. □

PROOF The algorithm follows the definition as described previously in steps 1–4. ■

5 The CSSA form

A program in SSA form has the property that each 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, a ϕ function is introduced to resolve the ambiguity. The ϕ function merges all the incoming reaching definitions to create a new definition for the variable [6]. In a parallel program, the single assignment property is disrupted by the presence of concurrent definitions to the variable. The CSSA framework solves this ambiguity with π functions. Each π function has $n + 1$ arguments; the unique incoming control flow edge and the n incoming conflict edges.

This section describes the algorithms needed to build the CSSA form as described in [15]. Algorithm 5.1 computes the CSSA form of a program. The algorithms to place ϕ functions and build factored use-def chains compute the sequential SSA form as described in [24]. Note that all the algorithms in this section are unmodified versions of the original references. They are only included to simplify the discussion of the complexity analysis of the CSSAME algorithm.

5.1 Computing guaranteed partial execution ordering

For each node n in the CCFG of the program, Algorithm 5.2 computes $prec(n)$, the set of nodes guaranteed to execute before n. This information is used when placing π functions to avoid adding an argument when two nodes are guaranteed to execute in a specific order.

Algorithm 5.1 CSSA algorithm.

INPUT: An explicitly parallel program P and its CCFG
OUTPUT: The program P in CSSA form

```
1: Find guaranteed execution ordering using Algorithm 5.2.
2: Build sequential SSA form using Algorithms 5.3 and 5.4.
3: Place $\pi$ functions using Algorithm 5.5.
```

Algorithm 5.2 Guaranteed Partial Execution Ordering.

INPUT: A Parallel Flow Graph $G = \langle N, E, Entry_G, Exit_G \rangle$
OUTPUT: $prec(n)$ for each node $n \in N$

```
1: /* Fold loop bodies into a representative node. */
2: /* $Loop(n)$ is a function that returns the set of nodes in a loop whose header is $n$. */
3: Build a sub-graph of $G$ such that:
   $N' \leftarrow N - \{n : m, n \in N \wedge n \in Loop(m) \wedge m$ is a loop header $\wedge m \neq n\}$
   $E' \leftarrow (E_{ct} \cup E_{dsync}) - \{(m, n) : m, n \in N \wedge (m \notin N' \vee n \notin N')\}$
4: foreach $n \in N'$ do
5:     $prec(n) \leftarrow \emptyset$
6: end for
7: Initialize work queue $Q$ with the successors of $Entry_G$
8: while $Q \neq \emptyset$ do
9:     Remove some node $n$ from $Q$
10:    $prec_{old} \leftarrow prec(n)$
11:    if $n$ is coend then
12:        $prec_{ct}(n) \leftarrow \bigcup_{(m,n) \in E_{ct}} prec(m) \cup \{n\}$
13:    else
14:        $prec_{ct}(n) \leftarrow \bigcap_{(m,n) \in E_{ct}} prec(m) \cup \{n\}$
15:    end if
16:    $prec_{sy} \leftarrow \bigcap_{(m,n) \in E_{dsync}} prec(m) \cup \{n\}$
17:    $prec(n) \leftarrow prec_{ct}(n) \cup prec_{sy}(n)$
18:    if $prec_{old} \neq prec(n)$ then
19:        Put control flow and synchronization successors of $n$ in $Q$
20:    end if
21: end while
22: foreach $n \in N - N'$ do
23:    /* $header(n)$ is a function that returns the header node */
24:    /* of the outermost loop enclosing $n$ */
25:    $prec(n) \leftarrow prec(header(n))$
26: end for
```

5.2 Computing the sequential SSA form

The CSSA algorithm calls for the computation of the sequential SSA form for the program. We compute the sequential SSA form using factored use-def chains [24]. Algorithm 5.3 adds ϕ functions to the graph and Algorithm 5.4 builds the use-def chains that link every variable use to its unique control reaching definition. These algorithms assume the existence of the following data structures:

1. $child(n)$ is the set of dominator children for node n.
2. $succ(n)$ is the set of successors of node n.
3. $whichPred(n \rightarrow m)$ is an index telling which predecessor of m corresponds to the control edge from n.
4. $DF(n)$ is the dominance frontier for node $n \in G$.
5. $D(v)$ is the set of nodes in G that contain a definition for variable v.
6. $Symbols$ is the set of variables used in the program.

Algorithm 5.3 Place ϕ functions.

INPUT: A Parallel Flow Graph $G = \langle N, E, Entry_G, Exit_G \rangle$
OUTPUT: Graph G with ϕ functions added at join nodes

```
1: foreach $n \in N$ do
2:   $inWork(n) \leftarrow \perp$
3:   $added(n) \leftarrow \perp$
4: end for
5: $workList \leftarrow \emptyset$
6: foreach $v \in Symbols$ do
7:   foreach $n \in D(v)$ do
8:     $workList \leftarrow workList \cup \{n\}$
9:     $inWork(n) \leftarrow v$
10:   end for
11:   while $workList \neq \emptyset$ do
12:     Remove some node $n$ from $workList$
13:     foreach $w \in DF(n)$ do
14:       if $added(w) \neq v$ then
15:         Add $\phi$ function for $v$ at $w$
16:         $added(w) \leftarrow v$
17:         if $inWork(w) \neq v$ then
18:           $workList \leftarrow workList \cup \{w\}$
19:           $inWork(w) = v$
20:         end if
21:       end if
22:     end for
23:   end while
24: end for
```

Algorithm 5.4 Build FUD chains.

INPUT: A Parallel Flow Graph $G = \langle N, E, Entry_G, Exit_G \rangle$ with ϕ functions added
OUTPUT: The graph with factored use-def chains

```
1: foreach $v \in Symbols$ do
2:   $currDef(v) \leftarrow \perp$
3: end for
4: call $search(Entry_G)$

5: procedure $search(x)$
6: foreach variable use or def or $\phi$ function $r \in x$ do
7:   $m \leftarrow$ variable referenced at $r$
8:   if $r$ is a use then
9:     $chain(r) \leftarrow currDef(m)$
10:   else if $r$ is a def or a $\phi$ function then
11:     $saveChain(r) \leftarrow currDef(m)$
12:     $currdef(m) \leftarrow r$
13:   end if
14: end for
15: foreach $y \in succ(x)$ do
16:   $j \leftarrow whichPred(x \rightarrow y)$
17:   foreach $\phi$ function $r$ in $y$ do
18:     $m \leftarrow$ variable referenced at $r$
19:     $\phi - chain(r)[j] \leftarrow currDef(m)$
20:   end for
21: end for
22: foreach $y \in child(x)$ do
23:   call $search(y)$
24: end for
25: foreach variable use or def or $\phi$ function $r \in x$ in reverse order do
26:   $m \leftarrow$ variable referenced at $r$
27:   if $r$ is a def or a $\phi$ function then
28:     $currDef(m) \leftarrow saveChain(r)$
29:   end if
30: end for
```

5.3 Placing π functions

The final phase of the CSSA algorithm traverses the graph placing π functions at every node that contains one or more conflicting variable uses. Algorithm 5.5 adds the required π functions to the graph.

Algorithm 5.5 Place π functions.

```
INPUT:   A Parallel Flow Graph $G = \langle N, E, Entry_G, Exit_G \rangle$ with FUD chains
OUTPUT:  The graph $G$ with $\pi$ functions added

1:  foreach $b \in N$ do
2:    foreach DU conflict edge $e = (a, b)$ do
3:      $v \leftarrow$ variable defined in $a$
4:      if $b$ does not have a $\pi$ function for $v$ then
5:        Insert a new $\pi$ function for $v$ in $b$
6:        $u \leftarrow$ conflicting use of $v$ in $b$
7:        $\pi(v)[0] \leftarrow chain(u)$
8:      end if
9:      if $n \notin prec(s)$ then
10:       $d \leftarrow$ conflicting def of $v$ in $s$
11:       append $d$ to $\pi(v)$
12:     end if
13:   end for
14: end for
```

5.4 Time complexity of the CSSA algorithm

The computation of the CSSA form is done in three phases. The first phase computes guaranteed partial execution ordering for all the nodes in the graph (Algorithm 5.2. In the worst case, every node will have to be compared to every other node in the graph. Hence, computing partial orderings can be done in $O(|N|^2)$.

The second phase computes the sequential SSA form for the program (Algorithms 5.3 and 5.4). This algorithm computes the SSA form in $O(r^3)$ time, where r is the maximum of the number of nodes ($|N|$), number of control edges ($|E_{ct}|$), number of assignments and number of variable references in the program [3, 6]. Note that it is possible to place ϕ terms using the linear time algorithms in [11] and [21]. We use the algorithms from [24] solely because they are easier to implement.

The third phase of the computation of the CSSA form places π functions at the concurrent join nodes of the graph [15]. By examining the π placing algorithm (Algorithm 5.5) we conclude that this phase can be computed in $O(|N|^2)$ time.

In conclusion, the CSSA form can be computed in $O(|N|^2)$ time when using the linear time algorithms for placing ϕ functions. If the traditional ϕ placing algorithms are used, then the CSSA form can be computed in $O(r^3)$ time. We shall use the latter bound for the remainder of this document.

6 The CSSAME form

Mutual exclusion analysis identifies memory interleavings that are not possible at runtime due to the synchronization structure of the program. This analysis allows the compiler to reduce the number of incoming conflict edges to nodes in the CCFG that use shared variables. This section describes our refinements to the CSSA framework [15]. We call this new form CSSAME (Concurrent SSA with Mutual Exclusion synchronization). While CSSA only recognizes `set`/`wait` synchronization, CSSAME extends it to include `lock`/`unlock` synchronization. Note that although we include lock variables in our analysis, for clarity of presentation we will not use SSA numbering for lock variables in the example programs. Since `lock` operations typically read and write the lock and `unlock` operations only write the lock, an implementation should create π functions for every `lock` node in the graph.

The key observation that gives rise to the CSSAME form is that π functions inside mutual exclusion sections might have one or more arguments for memory interleavings that cannot occur at runtime. We have developed two sufficient conditions, called *consecutive kills* and *protected uses*, for the removal of arguments from π functions inside mutex bodies. Both removal conditions can be easily implemented as predicates called by the compiler when analyzing mutex bodies.

6.1 Consecutive kills

If a variable is defined more than once inside a mutex body b, the only definitions that can be observed by other mutex bodies (in the same mutex structure) are those that reach the exit node of b. This is because all the mutex bodies in the same mutex structure are serialized and execute atomically. This situation is illustrated in Figure 4(a).

Definition 6.1 (reachability) Given a CCFG G, a definition D_v for a variable v *reaches* node $n \in G$ if there is a control path from the node containing D_v to n such that there is no other definition of v along that path [1]. $\Box$

Theorem 6.1 (consecutive kills) Let M_L be a mutex structure for lock variable L. Let D_a^B be a definition for a shared variable a inside a mutex body $B_L(n, x) \in M_L$. If D_a^B does not reach node x then D_a^B can be removed from all the π functions in any other mutex body $B'_L(n', x') \in M_L$ that have D_a^B as an argument. $\Box$

PROOF Let $U_a^{B'}$ be any use of a in $B'_L(n', x')$. Let d be the node containing D_a^B. Let u be the node containing $U_a^{B'}$. Since d and u are inside mutex bodies in the same mutex structure they cannot execute concurrently. Therefore, for every execution of the program that includes both mutex bodies there can only be two possible partial orderings between them:

1. $B_L(n, x)$ executes to completion before $B'_L(n', x')$. Even though node d executes

```
cobegin
  T0: begin
    lock(L);
    a1 = ...
    ...
    a2 = ...
    unlock(L);
  end

  T1: begin
    lock(L);
    ...
    /* Definition a1 cannot */
    /* reach this use. */
    a3 = π(a0, a1, a2); ⇒ a3 = π(a0, a2);
    ... = a3;
    unlock(L);
  end
coend
```

(a) Consecutive kills

```
cobegin
  T0: begin
    lock(L);
    ...
    a1 = ...
    /* Definition a1 protects further */
    /* uses of a in this mutex body. */
    a3 = π(a1, a2); ⇒ a3 = π(a1);
    ... = a3;
    unlock(L);
  end

  T1: begin
    lock(L);
    ...
    a2 = ...
    unlock(L);
  end
coend
```

(b) Protected uses

Figure 4: Removing memory conflicts.

before node u, the definition D_a^B cannot reach $U_a^{B'}$ because it is always killed by some other definition before it reaches the exit node of $B_L(n, x)$.

2. $B'_L(n', x')$ executes to completion before $B_L(n, x)$. Node u executes before node d, therefore D_a^B cannot reach $U_a^{B'}$.

Since it is impossible for the definition D_a^B to reach the use $U_a^{B'}$ then the argument representing D_a^B for the π function in $U_a^{B'}$ is not necessary. Therefore, it can be safely removed and the `DU(a)` conflict edge between d and u can be eliminated from the CCFG. ■

6.2 Protected uses

The second conflict removal opportunity is for uses that cannot be affected by definitions in other mutex bodies because they are protected by a local definition. Suppose that a conflicting variable a is used inside a mutex body B but its control reaching definition is inside B (Figure 4(b)). Since a is defined inside the mutex body, definitions made in other mutex bodies are killed by the internal definition of a.

Definition 6.2 (upward exposure for mutex bodies) Given a mutex body B, a use U_v^B in B for a variable v is *upward-exposed* [1] from B if U_v^B may use a definition outside of B. □

Theorem 6.2 (protected uses) Let M_L be a mutex structure for lock variable L. Let U_a^B be a conflicting use for a shared variable a inside a mutex body $B_L(n, x) \in M_L$. If U_a^B is not upward-exposed from $B_L(n, x)$ then the arguments for the π function for a coming from any other mutex body $B_L'(n', x') \in M_L$ can be removed. $\square$

PROOF Let $D_a^{B'}$ be a definition for variable a in mutex body $B_L'(n', x')$. Let d be the node in $B_L'(n', x')$ that contains the definition $D_a^{B'}$. Let u be the node in mutex body $B_L(n, x)$ that contains the use U_a^B. Since d and u are inside mutex bodies in the same mutex structure they cannot execute concurrently. Therefore, for every execution of the program that includes both mutex bodies there can only be two possible partial orderings between them:

1. $B_L(n, x)$ executes to completion before $B_L'(n', x')$. This means that node u executes before node d, therefore $D_a^{B'}$ cannot reach U_a^B.

2. $B_L'(n', x')$ executes before $B_L(n, x)$. Since U_a^B is not upward-exposed from $B_L(n, x)$, any definitions of a made before $B_L(n, x)$ starts executing are guaranteed to be killed by some other definition inside $B_L(n, x)$. Therefore, $D_a^{B'}$ cannot reach U_a^B.

Since the definition $D_a^{B'}$ cannot reach the use U_a^B then the argument representing $D_a^{B'}$ for the π function in U_a^B is not necessary. Therefore, it can be safely removed and the `DU(a)` conflict edge between d and u can be eliminated from the CCFG. $\blacksquare$

6.3 Modifying π functions inside mutex bodies

Using the properties of consecutive kills and protected uses inside mutex bodies, we can now examine every mutex body of the program trying to remove arguments from each of its π functions. Algorithm 6.1 traverses all the mutex bodies in the graph looking for π functions to rewrite. There are three main steps to the algorithm:

1. Lines 1–6 traverse all the mutex bodies in the program. For each mutex body b, it invokes the analysis routine in lines 7–27.

2. Lines 9–20 analyze all the π functions inside a mutex body b. For each π function, each of its arguments d is analyzed for compliance with Theorems 6.1 and 6.2.

   Checking for protected uses is a simple matter of checking whether the control reaching definition for the π function is dominated by the mutex body's entry node (since dominance and post-dominance information is already computed and stored in bitvectors, this can be computed in essentially constant time).

   Checking for consecutive kills can be done in $O(|confdefs|^2)$ time, where the value $|confdefs|$ represents the number of conflicting definitions made in the program. To check if a definition d reaches the exit node of a mutex body we traverse the post-dominator tree for d looking for a definition that post-dominates d and is post-dominated by the mutex body's exit node (i.e., we check whether there is another definition d' in the path from d to the exit node that kills d).

3. Lines 21–25 remove any π functions with no arguments for conflicting references.

Examining the nesting structure of the π rewriting algorithm we conclude that the total time complexity of the algorithm is $O(m \times mb \times mbsz \times |\pi| \times |confdefs|^2)$, were m is the number of lock variables in the program, mb is the total number of mutex bodies in the program, $mbsz$ is the average number of nodes that each mutex body contains, $|\pi|$ is the number of π functions in the program and $|confdefs|$ is the number of conflicting definitions in the program. A worst case scenario with a conflicting definition in every node and a conflicting use in every node will yield a time complexity of $O(|N|^3)$.

Lemma 6.1 (correctness of the π rewriting algorithm) The only arguments from π functions removed by Algorithm 6.1 represent memory interleavings that cannot occur at runtime. □

PROOF The algorithm only examines π functions inside mutex bodies. For each π function found it checks all the arguments that come from other mutex bodies in the same mutex structure. These are the only potential candidates for removal because they represent memory references protected by the same lock (line 15).

If d complies with one of the two sufficient conditions given by Theorems 6.1 and 6.2 then it may be safely removed because the definition represented by d cannot reach that particular use.

Finally, if after this analysis is done a π function p contains exactly one argument, it must be the argument for the incoming control edge to the node because this is the only argument that is never removed by Algorithm 6.1. Hence, this π function p can be removed from the graph. Before removing p, the algorithm updates the use-def pointer of the use affected by p (*chain(u)*) so that it points to p's control reaching definition (line 23). ■

6.4 Computing the CSSAME form

Algorithm 6.2 transforms an explicitly parallel program P to its CSSAME form. The algorithm is a direct extension of the CSSA algorithm [15]. Steps 2 and 4 incorporate the modifications needed to handle mutual exclusion synchronization.

The algorithm starts by building the concurrent control flow graph for P using the algorithms described in Section 4.2. Once the CCFG has been built, the algorithm creates the mutex structures for the mutual exclusion synchronization used in the program. The next step builds the CSSA form using the algorithms described in Section 5. Once the CSSA form has been computed, π functions are modified using Algorithm 6.1.

Algorithm 6.1 Rewrite π functions.

INPUT: A CCFG $G = \langle N, E, Entry_G, Exit_G \rangle$ in CSSA form
OUTPUT: The graph G in CSSAME form

```
1: /* Traverse all the mutex bodies in the graph looking for $\pi$ functions to rewrite. */
2: foreach lock variable $L_i$ do
3:   foreach mutex body $b \in MutexStruct(L_i)$ do
4:     call $rewrite(b)$
5:   end for
6: end for

7: /* Examine all the $\pi$ functions in $b$. */
8: procedure $rewrite(b)$
9: foreach node $n \in b$ do
10:   foreach $\pi$ function $p \in n$ do
11:     $v$ is the variable referenced by $p$
12:     /* If an argument of the $\pi$ function $p$ complies with Theorems 6.1 or 6.2, */
13:     /* then we may safely remove the argument from $p$ function. */
14:     foreach argument $d$ of $p$ coming from a conflict edge do
15:       if $d$ comes from another mutex body $b' \in MutexStruct(b)$ then
16:         if (the use of $v$ is not upward-exposed from $b$) or ($d$ does not reach the exit node of $b'$) then
17:           remove $d$ from $p$
18:         end if
19:       end if
20:     end for
21:     /* If $p$ is left with only one argument, remove $p$. */
22:     if $p$ has only one argument then
23:       $chain(u) \leftarrow$ first argument of $p$
24:       remove $p$ from $n$
25:     end if
26:   end for
27: end for
```

Algorithm 6.2 CSSAME algorithm.

INPUT: An explicitly parallel program P
OUTPUT: The program P in CSSAME form

```
1: Build the CCFG $G$ for $P$ using Algorithm 4.1.
2: Identify mutex structures using Algorithm 4.5.
3: Compute the CSSA form for the graph using Algorithm 5.1.
4: Rewrite $\pi$ functions using Algorithm 6.1.
```

Theorem 6.3 (correctness of the CSSAME algorithm) A program in CSSAME form is also in CSSA form and retains the single assignment property: every use is reached by exactly one definition. □

PROOF The CSSAME form is a direct extension of the CSSA form. The computation of the CSSA form is done using existing algorithms known to be correct [14, 24]. Lemma 6.1 proves that the only transformation done to the underlying CSSA form does not alter the single assignment property. Therefore, a program in CSSAME form is also in CSSA form and retains the single assignment property. ■

6.5 Time complexity of the CSSAME algorithm

Computing the CSSAME form does not increase the complexity of the CSSA algorithm significantly. The two major modifications to the original algorithm are steps 2 (computation of mutex structures) and 4 (rewriting of π functions). As discussed in Section 4, the identification of mutex structures can be done in $O(|E_{ct}|)$ time. The CSSA form is computed in $O(r^3)$ time, where r is the maximum of the number of nodes ($|N|$), number of control edges ($|E_{ct}|$), number of assignments and number of variable references in the program (Section 5.4). Finally, rewriting π functions can be done in $O(|N|^3)$ time. Therefore, the CSSAME algorithm has a worst time complexity of $O(|N|^3)$.

7 Optimizing explicitly parallel programs

Using the CSSAME form, new optimizations opportunities are now available. This section describes eight optimization techniques. The first two are adaptations of well-known sequential optimizations: constant propagation (Section 7.1) and dead code elimination (Section 7.2). The other six are new optimizations specifically designed for explicitly parallel programs: lock independent code motion (Section 7.3), mutex body localization (Section 7.4), single-writer multiple-readers code motion (Section 7.4.1), code sinking (Section 7.5), lock picking (Section 7.6) and lock partitioning (Section 7.7).

7.1 Constant propagation

Lee *et al.* [15] adapted the sequential Sparse Conditional Constant propagation (SCC) algorithm [23] to work with explicitly parallel programs; Concurrent Sparse Conditional Constant propagation (CSCC). We will use the program in Figure 2 to show how our extensions to the original CSSA framework can be used to improve the constant propagation algorithm when mutual exclusion is taken into account. Figure 5(a) is the original CSSA form without mutual exclusion extensions. Figure 5(b) shows the CSSAME form built using the algorithms in Section 6. Notice that the CSSAME form has fewer π functions than the CSSA form.

We now apply the CSCC algorithm to both the original CSSA form and the new CSSAME form. Notice that since CSSA does not recognize the mutual exclusion semantics of the program, the constant propagation algorithm cannot propagate any constants. On the other hand, translating the program to CSSAME allows the compiler to remove all the π functions for variable a in thread T_0. The key factor that allows the compiler to do this optimization is the assignment to variable a in thread T_0 immediately after the lock operation. Since all the statements in thread T_0 execute indivisibly, uses of variable a after the first assignment cannot possibly be affected by definitions of a made by thread T_1. This allows the compiler to propagate constants inside thread T_0 as if it were a sequential program. Figure 5(c) shows the results of applying the CSCC algorithm using CSSAME. Notice that we also include the results of the constant folding and unreachable code elimination. Both passes are possible using information gathered by the constant propagation algorithm [23]. Since we have not modified the CSCC algorithm, the optimizations performed are still correct as proved in [15].

7.2 Concurrent dead code elimination

Dead code refers to program statements that have no effect on the program output [6]. Although it is not common for the programmer to introduce dead code intentionally, dead code may be generated by optimizing transformations [1]. We introduce the Concurrent Dead Code Elimination algorithm (CDCE), an extension of the dead code elimination algorithm proposed by Cytron *et al.* [6] to work on explicitly parallel programs. The algorithm starts by marking as dead all the statements of the program except those that are assumed to affect the program output such as I/O statements or assignments to variables outside the current scope. This initial set of live statements is used to seed the work list maintained by the algorithm. The list is updated with every new statement that is marked live. When the list empties, all the statements still marked dead are removed from the program. A statement will be marked live if it satisfies one of the following conditions [6]:

1. The statement is assumed to affect the program output. Examples include I/O statements, calls to procedures that may have side effects, etc.

2. The statement contains a definition that reaches a use in a statement already marked as live.

3. The statement is a conditional branch and there is a live statement that is control dependent on this conditional branch.

```
$a_1$ = 0;
$b_1$ = 0;
cobegin
  $T_0$: begin
    lock(L);
    $a_2$ = 5;
    $a_3 = \pi(a_2, a_6)$;
    $b_2$ = $a_3$ + 3;
    if ($b_2 > 4$) {
      $a_4 = \pi(a_2, a_6)$;
      $a_5$ = $a_4$ + $b_2$;
    }
    $a_7 = \phi(a_2, a_5)$;
    $a_8 = \pi(a_7, a_6)$;
    $x_1$ = $a_8$;
    unlock(L);
  end

  $T_1$: begin
    lock(L);
    $b_3 = \pi(b_1, b_2)$;
    $a_6$ = $b_3$ + 6;
    $a_9 = \pi(a_6, a_2, a_5)$;
    $y_1$ = $a_9$;
    unlock(L);
  end
coend
$a_{10} = \phi(a_7, a_6)$;
print($x_1$, $y_1$);
```

(a) CSSA form

```
$a_1$ = 0;
$b_1$ = 0;
cobegin
  $T_0$: begin
    lock(L);
    $a_2$ = 5;

    $b_2$ = $a_2$ + 3;
    if ($b_2 > 4$) {

      $a_3$ = $a_2$ + $b_2$;
    }
    $a_4 = \phi(a_2, a_3)$;

    $x_1$ = $a_4$;
    unlock(L);
  end

  $T_1$: begin
    lock(L);
    $b_3 = \pi(b_1, b_2)$;
    $a_5$ = $b_3$ + 6;

    $y_1$ = $a_5$;
    unlock(L);
  end
coend
$a_6 = \phi(a_4, a_5)$;
print($x_1$, $y_1$);
```

(b) CSSAME form

```
$a_1$ = 0;
$b_1$ = 0;
cobegin
  $T_0$: begin
    lock(L);
    $a_2$ = 5;

    $b_2$ = 8;


    $a_3$ = 13;

    $a_4$ = 13;

    $x_1$ = 13;
    unlock(L);
  end

  $T_1$: begin
    lock(L);
    $b_3 = \pi(b_1, b_2)$;
    $a_5$ = $b_3$ + 6;

    $y_1$ = $a_5$;
    unlock(L);
  end
coend
$a_6 = \phi(a_4, a_5)$;
print($x_1$, $y_1$);
```

(c) Constant propagation using the CSSAME form

Figure 5: Constant propagation for the program in Figure 2.

The CDCE algorithm makes two modifications to the sequential algorithm:

- Condition 2 of Cytron *et al.*'s algorithm calls for the computation of reaching definition information for each live statement of the program. The rationale is that if statement s is live then any other statement that makes definitions with reached uses in s must also be marked live. We incorporate reaching definition and reached uses information in our CSSAME framework. We have adapted the corresponding sequential algorithms [24] by incorporating additional tests for π functions when traversing the SSA use-def chains.

- A `cobegin` statement will be marked live if there is at least one statement in two or more of its threads marked live. If the transformation leaves only one thread with live statements, the `cobegin`/`coend` construct will be replaced by the sequential code corresponding to the live thread. Serializing this live thread will cause all the synchronization operations in the thread to become dead. Hence, they can be safely removed.

These modifications to the sequential DCE algorithm are necessary to account for the concurrent activity in the program. Since reaching definition and reached uses information will be computed using both π and ϕ functions, a live use u in one thread, will keep concurrent definitions that reach u alive. Furthermore, the reduction of dependencies made possible by CSSAME directly benefits the elimination of dead code in the program. Most notably, the detection of consecutive kills inside a mutex body (Theorem 6.1) will help the detection of dead code inside mutex bodies.

To show the effects of CDCE, consider the program in Figure 2 after constant propagation has been performed (Figure 5(c)). As can be seen in the example program, all the assignments to variable a in T_0 are dead because they do not affect the output of the program (i.e., they do not reach any other use of a in the program). On the other hand, the assignment to b in T_0 cannot be considered dead because it is used by T_1. Note that a sequential dead code elimination algorithm would have erroneously marked the assignment to b dead because it lacks the appropriate reaching definition information. Figure 6(a) shows the result of a dead code pass on the code in Figure 5(c).

Theorem 7.1 The concurrent dead code elimination algorithm is correct. It only removes code that has no effect on program output. □

PROOF We will show that the CDCE algorithm does not mark dead statements that are really live. Since the sequential version is known to be conservative, we only need to consider the two modifications we have introduced.

Let D_v be a definition of variable v in thread T_0. Let U_v be a use of v in thread T_1. Assume that there is a conflict edge between the node containing D_v and the node holding U_v (i.e., the threads are concurrent and no synchronization prevents both memory operations from executing concurrently). Since the reaching definition information includes definitions reaching through conflict edges, if the statement holding U_v is marked live then the statement that contains D_v will also be marked live. The second condition is guaranteed by simply considering `cobegin`/`coend` structures as conditional branches.

■

The basic algorithm is the same, only the supporting data structures are different. In particular, for each statement s, $\mathit{Definers}(s)$ returns a set of statements that define variables used by s. In the sequential case, this corresponds to the set of reaching definitions of the variables used in s. In the parallel case, the computation of reaching definitions must also follow use-def chains through π functions.

Algorithm 7.1 Concurrent reaching definitions.

INPUT: A CCFG G in CSSAME form

OUTPUT: The set of reaching definitions for each variable used in the program and the set of reached uses for each variable defined in the program

```
/* $marked(d)$ is used to mark visited definitions */
/* $uses(d)$ is the set of uses reached by $d$ */
foreach variable definition $d$ in the program do
    $marked(d) \leftarrow \perp$
    $uses(d) \leftarrow \emptyset$
end for
foreach variable use $u$ in the program do
    $defs(u) \leftarrow \emptyset$
    call followChain(chain(u), u)
end for

/* Recursively follow use-def chains set up by the CSSAME algorithm */
procedure $followChain(d, u)$
if $marked(d) = u$ then
    return
end if
$marked(d) \leftarrow u$
/* If the reference $d$ is a definition, add it to the set of */
/* reaching definitions for $u$, and add $u$ to the set of reached uses of $d$ */
if $d$ is a definition for $u$ then
    Add $d$ to $defs(u)$
    Add $u$ to $uses(d)$
end if

/* If the reference $d$ is a $\phi$ or a $\pi$ function, follow the arguments */
if ($d$ is a $\phi$ function) or ($d$ is a $\pi$ function) then
    foreach function argument $j$ do
        call followChain($j$, $u$)
    end for
end if
```

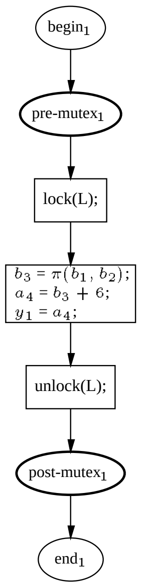
7.3 Lock independent code motion

Because of the sequential semantics imposed by mutual synchronization operations, it is desirable to minimize the time spent inside mutex bodies in the program. To achieve this

```
b1 = 0;
cobegin
  T0: begin
    lock(L);
    b2 = 8;
    x1 = 13;
    unlock(L);
  end

  T1: begin
    lock(L);
    b3 = π(b1, b2);
    a4 = b3 + 6;
    y1 = a4;
    unlock(L);
  end
coend
print(x1, y1);
```

(a) CDCE for program in Figure 5(c).



(b) Location of pre and post-mutex nodes for thread T_1.

```
b1 = 0;
cobegin
  T0: begin
    x1 = 13;
    lock(L);
    b2 = 8;
    unlock(L);
  end

  T1: begin
    lock(L);
    b3 = π(b1, b2);
    a1 = b3 + 6;
    unlock(L);
    y1 = a1;
  end
coend
print(x1, y1);
```

(c) Program from Figure 6(a) after LICM.

Figure 6: Concurrent Dead Code Elimination and Lock Independent Code Motion.

goal we can optimize the code inside mutex bodies as much as possible. Alternatively, we can minimize the amount of code executed inside a mutex body by moving code that does not need to be locked outside the mutex body. In this section we will explore ways of performing safe code motion on mutex bodies.

To determine what code can be safely moved outside a mutex body we must find those interior statements that are not affected by the presence of the lock. We call these *lock independent* statements. This is similar to the concept of loop-invariant code for standard loop optimization techniques [1]. However, the conditions that make a statement lock independent are different than those that make it loop invariant. Loop invariant computations are basically unaliased assignments with all their operands constant or with reaching definitions outside the loop. Lock independent code computes the same result whether it is inside a mutex body or not. For instance, a statement 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 the statement references variables not modified by any other concurrent thread in the program.

Definition 7.1 (lock independence) A statement inside a mutex body is *lock independent* if the variables it references cannot be modified concurrently[3]. □

Although lock independence is a necessary condition for code motion, it is not sufficient because the motion should also preserve all the control and data dependencies for

[3]Under some conditions it is possible to relax this restriction to allow concurrent modifications.

the statement. For instance, if the statement is inside a loop it cannot be moved out unless it is also loop invariant. To perform code motion we need to modify the flow graph to add two special nodes that will act as landing pads for statements moved out of each mutex body $B_L(n, x)$. We call these two nodes the *pre-mutex* and *post-mutex* node. The *pre-mutex* node is placed as an immediate strict dominator of n, while the *post-mutex* node is placed as an immediate strict post-dominator of x (Figure 6(b)). Theorem 7.2 provides a sufficient condition for moving statements outside mutex bodies.

Theorem 7.2 (lock independent code motion) Let s be a statement inside a mutex body $B_L(n, x)$. Let U_s be the set of variables used by s. Let D_s be the set of variables defined by s. Let a be the node containing s.

1. If (1) s is lock independent, (2) a dominates x, and (3) no statement between the `lock` statement in n and s contains a data dependency with variables in $D_s \bigcup U_s$, then s can be moved to the pre-mutex node of $B_L(n, x)$.

2. If (1) s is lock independent, (2) a dominates x, and (3) no statement between s and the `unlock` statement in x contains a data dependency with variables in $D_s \bigcup U_s$, then s can be moved to the post-mutex node of $B_L(n, x)$. □

PROOF

1. To prove that it is safe to move s to the pre-mutex node of B we must determine whether the data and control dependencies in s will be affected by the motion:

   (1) Since s is lock independent, no variable in D_s and U_s is in conflict. Therefore, moving s would not introduce memory conflicts.

   (2) Since a dominates the exit node x, moving s will not modify any control dependencies because s executes for every execution of the mutex body.

   (3) Finally, since no statement between `lock` and s contains a data dependency with variables in $D_s \bigcup U_s$, moving s to the pre-mutex node will not introduce new data dependencies nor modify existing data dependencies in the program.

   Given that none of the data and control dependencies of s are affected by the motion, we conclude that it is safe to move s to the pre-mutex node of B.

2. Proving the second part of the theorem is similar. In particular, conditions (1) and (2) are identical so they will not be repeated here.

   (3) Since no statement between s and `unlock` contains a data dependency with variables in $D_s \bigcup U_s$, moving s to the post-mutex node will not introduce new data dependencies nor modify existing data dependencies in the program.

   Given that none of the data and control dependencies of s are affected by the motion, we conclude that it is safe to move s to the post-mutex node of B. ■

Algorithm 7.2 Lock independent code motion (LICM).

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

OUTPUT: The graph with lock independent code moved to the corresponding pre-mutex and post-mutex nodes

```
1: foreach lock variable $L_i$ do
2:   foreach mutex body $B_{L_i}(n, x) \in MutexStruct(L_i)$ do
3:     /* First do a forward search looking for statements to move to the pre-mutex node. */
4:     /* This phase might also find statements that can be moved to the post-mutex node. */
5:     foreach statement $s \in B_{L_i}(n, x)$ in forward order do
6:       if $isMovable(s, B_{L_i}(n, x))$ then
7:         if $premutex(s)$ then
8:           move $s$ to the pre-mutex node
9:         else
10:          move $s$ to the post-mutex node
11:        end if
12:      end if
13:    end for
14:    /* Now do a backward search looking for statements to move to the post-mutex node. */
15:    /* This is needed because the previous pass might not have moved some down-movable */
16:    /* statements that were blocking each other. */
17:    foreach statement $s \in B_{L_i}(n, x)$ in reverse order do
18:      if $isMovable(s, B_{L_i}(n, x))$ then
19:        /* Notice that this phase can *only* find down-movable statements. */
20:        move $s$ to the post-mutex node
21:      end if
22:    end for
23:    /* If the mutex body is empty then remove the locking operations. */
24:    if $SDOM^{-1}(n) \bigcap SPDOM^{-1}(x) = \emptyset$ then
25:      remove $n$ and $x$ from the graph
26:    end if
27:  end for
28: end for
29: return
30: function $isMovable(s, B_L(n, x))$
31: $movable(s) \leftarrow$ FALSE
32: $premutex(s) \leftarrow$ FALSE
33: $postmutex(s) \leftarrow$ FALSE
34: $a \leftarrow$ node containing $s$
35: /* Only consider lock independent statements in nodes that dominate $x$ */
36: if $a\ DOM x$ and $s$ is lock independent then
37:   $U_s \leftarrow$ variables used in $s$
38:   $D_s \leftarrow$ variables defined in $s$
39:   /* Check if $s$ can be moved to the pre-mutex node. */
40:   if no statement between lock(L) and $s$ has a dependency with variables in $D_s \bigcup U_s$ then
41:     $movable(s) \leftarrow$ TRUE
42:     $premutex(s) \leftarrow$ TRUE
43:   end if
44:   /* Check if $s$ can be moved to the post-mutex node. */
45:   if no statement between $s$ and unlock(L) and $s$ has a dependency with variables in $D_s \bigcup U_s$ then
46:     $movable(s) \leftarrow$ TRUE
47:     $postmutex(s) \leftarrow$ TRUE
48:   end if
49: end if
50: return $movable(s)$
51: end
```

Algorithm 7.2 implements the concepts described previously. The algorithm makes two passes over all the statements inside a mutex body. The first pass (lines 10–20) traverses all the statements starting at the `lock(L)` operation trying to move statements to the pre-mutex node. Each statement is analyzed by the function $isMovable$ (lines 30–51) which determines whether the given statement complies with the LICM requirements (Theorem 7.2). Since each statement has to be compared with all its predecessors and successors in the mutex body, this phase can be performed in $O(|S|^2)$ time (S is the set of all statements in the program).

Notice that it might be possible that a statement can be moved to both the pre-mutex and the post-mutex nodes. In that case a cost model should determine which node is more convenient. We will base our cost model on the effects of lock contention. Suppose that there is high contention on a particular lock. All the statements moved to the pre-mutex node will not be affected by it because they execute before the acquisition of the lock. However, statements moved to the post-mutex 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 pre-mutex and post-mutex nodes, the pre-mutex node is selected.

The second pass (lines 21–29) does a backward traversal of the statements in the mutex body to circumvent the following problem: suppose that two statements s_1 and s_2 can be moved to the post-mutex node and s_1 is the predecessor to s_2 in a forward traversal. When the algorithm encounters s_1 it will conclude that it cannot be moved to the post-mutex node because s_2 is blocking it. However, since s_2 can be moved to the post-mutex node, s_1 should also be moved. This situation is avoided by doing a backward traversal on the mutex body. Similarly to the previous phase, this pass can be computed in $O(|S|^2)$ time.

Finally, if the mutex body is empty at the end of the transformation, the `lock` and `unlock` nodes are removed (lines 30–33). The total time complexity for the LICM algorithm is then $O(m \times mb \times (|S|^2 + |S|^2))$ which can be approximated to $O(m \times mb \times |S|^2)$. As before, we expect m (number of lock variables) and mb (number of mutex bodies in the program) to be relatively small compared to $|S|$.

Applying these conditions to the program in Figure 6(a) we obtain the equivalent program in Figure 6(c). Notice that both assignments to variables x and y can be safely moved out of each mutex body because there are no conflicting definitions in their sibling threads. Also, notice that even though the statement $x_1 = 13$ can be moved to both landing pads, it is more convenient to move it to the pre-mutex node.

7.3.1 Moving lock independent control structures

In this section we will consider an extension to the base LICM algorithm to allow whole control structures to be hoisted out of mutex bodies. The basic mechanism is straightforward. Before executing Algorithm 7.2 to perform code motion on individual statements, Algorithm 7.3 is executed to attempt moving control structures first.

The algorithm starts by identifying sub-graphs in G containing control structures (line 2). This computation is performed using standard interval analysis techniques [1]. Basically, control structures have similar properties as mutex bodies, they form a single-entry, single-exit region of the graph. An entry node dominates all the nodes in the control structure. An exit node post-dominates all the nodes in the control structure.

Once the sub-graphs have been identified, each sub-graph H whose header node ($head_H$) dominates the mutex body's exit node (x) is analyzed (Lines 4–25). The analysis process is basically the same as the base LICM algorithm with one notable difference. The 4 conditions in theorem 7.2 must be slightly modified to mask out the enclosing control structure. In particular, condition (2) is not necessary for the interior nodes of H, only the header node of H ($header_H$) should dominate the mutex body's exit node (x). Condition (3) is the same but it must not be checked against statements inside H. Otherwise, it could generate false positives that would impede moving H outside the mutex body.

The algorithm keeps two counters to keep track of the potential destinations for each statement. After analyzing all the statements in H the counters are compared against the total number of statements (Lines 26–44). A mismatch means that some statements cannot move in that particular direction. The sub-graph can only be moved if *all* the interior statements can be moved in the same direction. Finally, the same cost analysis used in the LICM algorithm is performed, if H can be moved in both directions, it will be hoisted to the pre-mutex node.

7.4 Mutex body localization

In this section we will discuss a transformation technique that may enhance the opportunities for further optimization of the program. Consider a mutex body B that modifies a shared variable a (Figure 7(a)). With the exception of the definition reaching the exit node of B, all the modifications done to a inside the mutex body can only be observed by the thread.

Given these conditions, it is possible to create a local copy of a and replace all the references to a inside the mutex body to references to the local copy (Figure 7(b)). We call this transformation *mutex body localization*. 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 upward-exposed use in the mutex body.

2. At the end of the mutex body, the shared copy is updated from the local copy of the variable if there is at least one definition of the variable inside the mutex body.

Algorithm 7.3 LICM for control structures.

INPUT: A CCFG $G = \langle N, E, Entry_G, Exit_G \rangle$ in CSSAME form
OUTPUT: The graph with lock independent control structures moved to the corresponding pre-mutex and post-mutex nodes

```
1: foreach lock variable $L_i$ do
2:   foreach mutex body $B_{L_i}(n, x) \in MutexStruct(L_i)$ do
3:     /* Build sub-graphs for all the control structures in the mutex body */
4:     build sub-graphs for $B_{L_i}(n, x)$
5:     /* Traverse sub-graphs checking if all the interior statements are group movable */
6:     foreach subgraph $H$ do
7:       if $head_H \; DOM x$ then
8:         $movable_H \leftarrow$ TRUE
9:         $stmts \leftarrow 0$
10:        foreach statement $s$ in $H$ do
11:          /* Use Theorem 7.2 to determine if $s$ is movable */
12:          /* Note that we alter the conditions slightly */
13:          /* to mask out the control structure containing $s$. */
14:          $stmts \leftarrow stmts + 1$
15:          call isMovable($s$)
16:          if $movable_s$ then
17:            /* Increment counters for potential destinations */
18:            if $premutex_s$ then
19:              $pre \leftarrow pre + 1$
20:            else
21:              $post \leftarrow post + 1$
22:            end if
23:          else
24:            /* If $s$ cannot move, then $H$ cannot move. */
25:            $movable_H \leftarrow$ FALSE
26:          end if
27:        end for
28:        if $movable_H$ then
29:          /* $H$ can only go to a landing pad if all the statements can. */
30:          /* Compare the destination counters to see if we can move $H$. */
31:          $premutex_H \leftarrow$ FALSE
32:          $postmutex_H \leftarrow$ FALSE
33:          if $pre = stmts$ then
34:            $premutex_H \leftarrow$ TRUE
35:          end if
36:          if $post = stmts$ then
37:            $postmutex_H \leftarrow$ TRUE
38:          end if
39:          if $premutex_H$ = FALSE and $postmutex_H$ = FALSE then
40:            $movable_H \leftarrow$ FALSE
41:          end if
42:          if $movable_H$ then
43:            move $H$ using the same analysis used in Algorithm 7.2
44:          end if
45:        end if
46:      end if
47:    end for
48:  end for
49: end for
```

```
lock(L);



a = 0;
while (a <= X) {
  a = a + Y;
}

unlock(L);
```

(a) A mutex body before localization

```
lock(L);
/* p_a is a local variable    */
/* generated by the compiler */
p_a = 0;
while (p_a <= X) {
  p_a = p_a + Y;
}
a = p_a;
unlock(L);
```

(b) After localization

Figure 7: Mutex body localization.

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 7.5 makes local copies of a variable a inside a mutex body $B_L(n, x)$ if the variable can be localized. To determine whether the variable a can be localized it calls Algorithm 7.4 which returns TRUE if a can be localized inside mutex body $B_L(n, x)$. The localization algorithm relies on two data structures that can be easily built during the π rewriting phase of the CSSAME algorithm (Algorithm 6.1):

1. $exposedUses(n)$ is the set of upward-exposed uses from the mutex body $B_L(n, x)$. This set is associated with the entry node n.

2. $reachingDefs(x)$ is the set of definitions that can reach the exit node x of $B_L(n, x)$. Notice that since the program is in CSSAME form this set is actually one definition per variable. If more than one definition of a variable can reach node x, they will be represented by a single ϕ function.

Algorithm 7.5 starts by checking whether the variable can be localized (lines 1–4). It then checks where are the local copies 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 the 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 (Algorithm 6.2). 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. 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 only 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 terms for a must be converted into ϕ terms 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 7.4 Localization test ($localizable$)

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

1:  $M_L \leftarrow$ mutex structure containing $B_L(n, x)$
2:  /* Check every conflicting reference $r$ to $a$ in the program. All the conflicting */
3:  /* references to $a$ must occur inside mutex bodies of $M_L$, otherwise $a$ is not localizable. */
4:  **foreach** conflicting reference $r \in Refs(a)$ **do**
5:    /* If we cannot find $r$ in any of the mutex bodies of $M_L$, then $a$ is not localizable. */
6:    $protected \leftarrow$ FALSE
7:    **foreach** mutex body $B'_L(n', x') \in M_L$ **do**
8:      **if** $n'$ $SDOM Node(r)$ **and** $x'$ $SPDOM Node(r)$) **then**
9:        $protected \leftarrow$ TRUE
10:     **end if**
11:   **end for**
12:   **if not** $protected$ **then**
13:     **return** FALSE
14:   **end if**
15: **end for**
16: /* All the references to $a$ are protected. Therefore, $a$ is localizable. */
17: **return** TRUE
```

Algorithm 7.5 Mutex body localization

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

```
/* Check if $a$ can be localized (Algorithm 7.4) */
if not $localizable(a, B_L(n, x))$ 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 (Algorithm 6.1). If there are */
/* upward-exposed uses of $a$ then we need to make a local copy of $a$ at the start of $B_L(n, x)$. */
$needEntryCopy \leftarrow$ 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 node of $B_L(n, x)$. */
/* Since the program is in CSSAME form, the definitions that reach the exit node $x$ */
/* have already been computed (Algorithm 6.1). 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 end 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 $B_L(n, x)$ do
    replace $a$ with $p\_a$
end for
update CSSAME information for all references to $p\_a$ inside $B_L(n, x)$
```

As mentioned before, this transformation by itself does not necessarily improve the performance of a program but it opens up new optimization opportunities. The main effect of localization is that it might uncover 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. It could also be possible to split the mutex body so that only the creation of the local copies and the updates to shared memory are protected by the lock. Some of these transformations need to be evaluated with appropriate cost models to determine their usefulness. In the following sections we will describe two related optimizations that take advantage of the effects of mutex body localization.

7.4.1 Single writer, multiple readers code motion

Suppose that a parallel program exhibits an access pattern to a shared variable v such that

1. v is read and written by exactly one thread T_w and it is read-only in all of the threads concurrent with T_w (i.e. there is a single writer and multiple readers for v), and

2. all the references to v are atomic with respect to the operation being performed (i.e., v is not an aggregate data type that may require multiple memory operations to update or retrieve).

We will also assume that variable v is accessed inside critical sections of the code (otherwise the optimization is clearly unnecessary). Under these circumstances it is possible to localize the references to v in T_w so that atomicity can be maintained without the requirement of locks.

For example, consider the program in Figure 8(a). Thread T_0 computes a value for a, checks a bound and updates a if necessary (assume that global variables X and Y have no conflicts). Both threads T_1 and T_2 read a but never modify it. The synchronization on a is necessary to prevent threads T_1 and T_2 from reading intermediate values of a while T_0 computes. Suppose that we localize variable a inside T_0 to obtain the equivalent program in 8(b). Since X and Y contain no conflicts and the references to a have been localized, all the statements inside the mutex body are now lock independent and can be moved out to obtain the program in Figure 8(c).

7.4.2 Relaxing lock independence requirements

The optimized program in Figure 8(c) contains no lock independent statements according to Definition 7.1. However, because of the memory semantics assumed in our model

```
X = ...
Y = ...
cobegin
  T_0: begin
    ...
    lock(L);
    a = 0;
    while (a <= X) {
      a = a + Y;
    }

    unlock(L);
  end

  T_1: begin
    lock(L);
    ... = a;
    unlock(L);
  end

  T_2: begin
    lock(L);
    ... = a;
    unlock(L);
  end
coend
```

(a) Original program

```
X = ...
Y = ...
cobegin
  T_0: begin
    ...
    lock(L);
    p_a = 0;
    while (p_a <= X) {
      p_a = p_a + Y;
    }
    a = p_a;
    unlock(L);
  end

  T_1: begin
    lock(L);
    ... = a;
    unlock(L);
  end

  T_2: begin
    lock(L);
    ... = a;
    unlock(L);
  end
coend
```

(b) After localization

```
X = ...
Y = ...
cobegin
  T_0: begin
    ...
    p_a = 0;
    while (p_a <= X) {
      p_a = p_a + Y;
    }
    lock(L);
    a = p_a;
    unlock(L);
  end

  T_1: begin
    lock(L);
    ... = a;
    unlock(L);
  end

  T_2: begin
    lock(L);
    ... = a;
    unlock(L);
  end
coend
```

(c) After LICM

```
X = ...
Y = ...
cobegin
  T_0: begin
    ...
    p_a = 0;
    while (p_a <= X) {
      p_a = p_a + Y;
    }

    a = p_a;

  end

  T_1: begin

    ... = a;

  end

  T_2: begin

    ... = a;

  end
coend
```

(d) After relaxing lock independence

Figure 8: Effects of localization in the presence of single writer, multiple readers patterns

we can mark as lock independent certain statements that do not fit the criteria for lock independence.

Suppose that the same conditions for single writer, multiple readers hold for shared variable v. Furthermore, we require that the writing thread writes to v only once (this is guaranteed by the localization process described previously). Under these circumstances, all the statements that reference v can be safely marked as lock independent. Using this relaxed criterion for lock independence we can further optimize the program in Figure 8(c) to remove all the locks that protect variable a and obtain the equivalent program in Figure 8(d).

7.5 Code sinking

Code sinking is a new code motion strategy designed to increase the granularity of individual threads and avoid the sequential processing overhead for threads that do not use computations made in sequential portions of the code. We will use a simple example to illustrate the idea. Consider the program in Figure 9(a). The first three lines of the

```
a = 5;
b = 4;
c = 2;
cobegin
  T0: begin
        t = a * b;
  end

  T1: begin
        v = c / 3;
  end
coend
print(t, v);
```

(a) Before code sinking

```
cobegin
  T0: begin
    a = 5;
    b = 4;
    t = a * b;
  end

  T1: begin
    c = 2;
    v = c / 3;
  end
coend
print(t, v);
```

(b) After code sinking

Figure 9: Code sinking optimization.

program compute new values for variables a, b and c. Thread T_0 uses variables a and b and thread T_1 only uses c. Figure 9(b) shows the results of applying the code sinking optimization to the program on the left. Since thread T_1 does not use variables a or b, both assignments in the sequential section of the program can be moved inside T_0 so that T_1 does not have to pay the sequential overhead for computations that it will not use. The same reasoning is applied to thread T_0 when moving the assignment of variable c to the body of thread T_1.

Similar to other code motion strategies, code sinking can only operate on a statement if all the original data and control dependencies are preserved after the move and no data races are introduced in the program. We will describe a set of sufficient requirements to guarantee the safety of this optimization. If the decision is made to sink a statement s into a thread T, s will be moved to a special node called a *landing pad* for T. The landing pad is inserted inside thread T so that its immediate dominator is $begin_T$.

Theorem 7.3 (code sinking) Let d be the only definition of variable v made by a statement s inside CCFG node a. Let u be a reached use for d inside CCFG node b in thread T. Let p be the landing pad for T. It is safe to move s into p if all the following conditions are met:
(1) $a\ DOM\ begin_T$,
(2) there are no dependencies between d and references to v in threads concurrent with T, and
(3) there are no dependencies between d and references to v along any path from a to $begin_T$. □

PROOF We will show that moving s into p under these conditions will not alter any data or control dependencies in the program. We start by examining each of the conditions independently:

(1) Since node a dominates node $begin_T$ then node a also dominates node b which means that a and b cannot execute concurrently. Therefore d reaches u via control edges (i.e., the references do not conflict). Moving s into p does not affect the data and control dependency between d and u because the landing pad p also dominates b.
(2) Since no other thread concurrent with T references v, moving s into p does not introduce any data races in the program.
(3) Finally, since there are no dependencies for d along any path from a to $begin_T$, moving d into p does not affect data dependencies with other references to d in the program.

Given that the three conditions guarantee that the movement does not affect any data or control dependencies and it does not introduce data races in the program, we conclude that the transformation is safe. ■

The previous conditions assume that statement s contains only one definition. This can be generalized to statements containing multiple definitions provided that all definitions meet the same requirements. Algorithm 7.6 performs code sinking on an explicitly parallel program. The algorithm examines all the non-conflicting uses in every thread T of the program. We are only interested in non-conflicting uses because conflicting uses will have reaching definitions from concurrent threads. For each non-conflicting use u found, the algorithm computes the set of reaching definitions for u. Each reaching definition is examined to determine whether it complies with the three conditions required by Theorem 7.3. The algorithm requires the following information:

- $Conflicts(r)$ is the set of references that conflict with reference r.
- $ReachingDefs(u)$ is the set of reaching definitions for reference u (Algorithm 7.1).
- $ReachedUses(d)$ is the set of uses reached by d (Algorithm 7.1).
- $Conc(a, b)$ returns TRUE if nodes a and b can execute concurrently (Algorithm 4.2).

7.6 Lock picking

It is sometimes possible to remove some synchronization instructions in the program without affecting its correctness. In this section we will describe techniques that allow the compiler to detect and remove superfluous `lock`/`unlock` operations in the program. We collectively refer to these techniques as *lock picking* strategies.

The success of these strategies largely depends on the quality of non-concurrency information that the compiler is able to gather from the program. For instance, the

Algorithm 7.6 Code sinking.

INPUT: A CCFG $G = \langle N, E, Entry_G, Exit_G \rangle$ in CSSAME form
OUTPUT: The graph with some statements moved to their corresponding *landing pad* nodes

```
1: foreach thread $T$ do
2:   foreach node $a \in T$ do
3:     /* Examine all the non-conflicting uses in $a$. */
4:     foreach use $u$ in $a$ such that $Conflicts(u) = \emptyset$ do
5:       /* Check if any of the reaching definitions for $u$ can be moved to $T$. */
6:       $RD \leftarrow ReachingDefs(u)$
7:       foreach reaching definition $rd \in RD$ do
8:         /* Check first condition. The node containing $rd$ should dominate $begin_T$. */
9:         $b \leftarrow$ node containing $rd$
10:        if $b$ $DOM$ $begin_T$ then
11:          hasConcurrentUses ← FALSE
12:          hasUsesBeforeThread ← FALSE
13:          $RU \leftarrow ReachedUses(rd)$
14:          foreach reached use $ru \in RU$ such that $ru \neq u$ do
15:            $b \leftarrow$ node containing $ru$
16:            /* Check second condition. The reached use $ru$ should not be in a */
17:            /* thread concurrent with $T$. */
18:            /* Algorithm 4.2 computes the concurrency relation. */
19:            if $Conc(a, b)$ = TRUE then
20:              hasConcurrentUses ← TRUE
21:            end if
22:            /* Check third condition. The reached use $ru$ should not be before */
23:            /* the begin node of $T$. */
24:            if $begin_T$ $PDOM$ $b$ then
25:              hasUsesBeforeThread ← TRUE
26:            end if
27:          end for
28:          if (not hasConcurrentUses) and (not hasUsesBeforeThread) then
29:            if $T$ does not have a landing pad node then
30:              insert a new landing pad for $T$
31:            end if
32:            move the statement containing $d$ to the landing pad
33:          end if
34:        end if
35:      end for
36:    end for
37:  end for
38: end for
```

```
cobegin
  T0: begin          T1: begin
      ...                ...
      lock(L);           Wait(a);
      ...                ...
      unlock(L);         lock(L);
      ...                ...
      Set(a);            unlock(L);
      end                  ...
                         end
                     coend
```

(a) Lock picking by precedence. Mutex body in T_0 is guaranteed to execute before mutex body in T_1.

```
cobegin
  T0: begin               T1: begin
      ...                     ...
      lock(X);                lock(X);
      if (a > b) {            if (a < b) {
        b = 3;                    b = 5 - a;
        lock(Y);                lock(Y);
        z = z + 5;                z = z - a;
        unlock(Y);              unlock(Y);
      }                        }
      unlock(X);              unlock(X);
      ...                     ...
  end                       end
                          coend
```

(b) Lock picking by inclusion. Mutex bodies for X always include mutex bodies for Y.

Figure 10: Two lock picking strategies.

compiler can use one of several techniques for computing guaranteed execution ordering [4, 15, 17] to determine if an ordering exists between mutex bodies in the same mutex structure. An algorithm for computing guaranteed execution ordering returns pairs of nodes a and b such that a precedes b on every execution of the program.

Theorem 7.4 (lock picking by precedence) Let M_L be a mutex structure with k mutex bodies $B_L^1(n_1, x_1)$, $B_L^2(n_2, x_2)$, ..., $B_L^k(n_k, x_k)$. If there exists a mutex body $B_L^i(n_i, x_i)$ such that $\forall j \neq i : x_j$ precedes n_i then the nodes n_i and x_i can be safely removed. □

PROOF If the exit nodes from all the sibling mutex bodies precede the entry node to mutex body $B_L^i(n_i, x_i)$ then all the mutex bodies in M_L will execute to completion before B_L^i starts executing. Therefore, it is not necessary for B_L^i to hold the lock L and both n_i and x_i can be removed. ■

Theorem 7.4 can be iteratively applied to all the mutex bodies in the same mutex structure until no more partial orders can be guaranteed between the entry node of one mutex body and all the exit nodes of its siblings. Consider the program in Figure 10(a). Thread T_1 will wait until T_0 sets event a before trying to acquire the lock L. Since T_0 sets a after it releases the lock L, it is not necessary for T_1 to lock L anymore. This gives rise to a second opportunity for lock picking: *orphaned mutex structures*. Assume that the locks in thread T_1 of Figure 10(a) have already been removed. Removing the locks in thread T_1 has left the mutex structure for lock L with only one mutex body, namely the one defined in T_0. This is clearly unnecessary and the mutex body can be safely removed. Notice that this analysis can only be applied to pure mutex structures, otherwise the compiler would remove locks that might be needed at runtime.

Another opportunity for lock picking is the presence of nested mutex bodies. The program in Figure 10(b) shows an example. Assume that both threads T_0 and T_1 are the only two threads in the program and can execute concurrently. Notice that all the mutex bodies for Y are nested inside mutex bodies for X. This makes the locking operations on Y unnecessary because X already protects the same references protected by Y.

Theorem 7.5 (lock picking by inclusion) Let M_{L_1} be a mutex structure for lock L_1. Let M_{L_2} be a mutex structure for lock L_2. If $\forall B^i_{L_1}(n_i, x_i) \in M_{L_1}\ \exists B^j_{L_2}(n_j, x_j) \in M_{L_2}$ such that $n_j \in B_{L_1}$ and $x_j \in B_{L_1}$, then all the entry and exit nodes for mutex bodies in M_{L_2} can be removed. □

PROOF Since n_j and x_j are included inside the mutex body $B^i_{L_1}(n_i, x_i)$, then acquiring lock L_2 implies that lock L_1 is already held by the calling thread. Furthermore, lock L_2 is always released before L_1. This has two implications. First, acquiring L_2 will always succeed because the access to L_2 is serialized by L_1. Second, lock L_1 is always held while mutex body B_{L_2} executes. Therefore, acquiring lock L_2 is not really necessary and the entry and exit nodes for mutex bodies in M_{L_2} can be removed. ■

7.7 Lock partitioning

Lock partitioning is a new optimization technique that examines all the mutex bodies in a single mutex structure to determine whether they access the same set of variables. Consider a program that uses a single lock L to serialize the access to variables a, b, x and y. Assume that only one mutex body references x and y while the other mutex bodies in the program reference a and b. We can safely replace L with two locks, one for the mutex body referencing x and y and another one for the mutex bodies referencing a and b. The key idea is that if the mutex bodies are accessing different sets of variables, then protecting all the references with a single lock is not necessary and restricts concurrency in the program. Lock partitioning will determine how many disjoint sets of variables are referenced by the different mutex bodies and replace the original lock with one lock for each set of variables. In the following discussion we assume that the entry nodes for all mutex bodies in the same mutex structure can execute concurrently. If the control or synchronization structure of the program prevents the entry nodes from executing concurrently then this analysis is clearly unnecessary. Therefore we shall assume that mutex structures have already been pruned by a lock picking pass (Section 7.6).

Theorem 7.6 (lock partition) Let M_L be a mutex structure with n mutex bodies B^1_L, B^2_L, …, B^n_L. Let V_i be the set of variables accessed by mutex body B^i_L. Let $V = \{V_1, V_2, \ldots, V_n\}$. If there exists a partition of V $P = \{P_1, P_2, \ldots, P_m\}$ such that

(1) $\forall P_i \in P : \forall V_k, V_l \in P_i : V_k \bigcap V_l \neq \emptyset$, and

(2) $\forall P_i, P_j \in P : \forall V_k \in P_i, V_l \in P_j : V_k \bigcap V_l = \emptyset$

then M_L can be partitioned into m mutex structures $M_{L_1}, M_{L_2}, \ldots M_{L_m}$. □

PROOF Each partition P_i in P contains all the sets in V that access a common set of variables (first condition). Furthermore, all the partitions in P access different variables (second condition). Therefore, it is not necessary to use the same lock to protect all the variables. Since each partition P_i references a common set of variables, all the mutex bodies corresponding to the sets of variables in each P_i can be protected by a new lock L_i. No data races will be introduced by this change because there are no conflicting references between the different partitions in P. ■

Algorithm 7.7 applies the conditions described in Theorem 7.6 to partition mutex structures in a program. The algorithm starts by collecting the variables referenced in each mutex body B_i. The main loop of the algorithm builds the partitions. Each mutex body B_i is compared to each of the existing partitions represented by the array $P[]$. If there exists a $P[j]$ such that $P[j]$ has variables in common with V_i, then all the variables in V_i are added to $P[j]$ and the mutex body B_i is added to the list of mutex bodies in $M[j]$. Otherwise, the number of partitions, m, is incremented, V_i is used to initialize $P[m]$ and B_i is used to initialize $M[m]$. At the end of this loop, if more than one partition was created, the algorithm creates m new locks and assigns them to each of the new mutex structures stored in $M[]$.

8 Applying the framework to real programs

The example programs presented in this paper are necessarily simple because they are only meant to illustrate the potential effect of each transformation. We do not expect experienced programmers to write code like the examples shown in the paper. However, we expect these techniques to be useful in a number of situations. For instance, consider a high-level programming language like Java. Due to the thread-safe characteristics of the Java libraries, application programs may spend up to half their execution time performing unnecessary synchronization [2]. 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. Techniques like the lock picking strategies or lock independent code motion will likely benefit these applications.

We also expect similar benefits for parallel programs generated via high-level programming environments. We are currently investigating the application of these techniques to a programming environment for object-oriented parallel programs based on design patterns. These tools must generate conservatively correct code based on code skeletons that might contain over-constrained synchronization. Similar to the previous case, machine generated code must be overly conservative for generality and safety. Since design patterns contain a significant amount of semantic information, the compiler can make a more informed decision when analyzing the code for memory conflicts and synchronization. Key to the success of these approaches is the ability of the compiler

Algorithm 7.7 Lock partitioning.

INPUT: A CCFG $G = \langle N, E, Entry_G, Exit_G \rangle$ in CSSAME form
OUTPUT: The graph with some mutex structures partitioned

```
1:  foreach lock variable $L$ do
2:    /* Determine variables referenced by each mutex body. */
3:    foreach mutex body $B_i \in M_L$ do
4:      $V_i \leftarrow$ set of variables referenced in $B_i$
5:    end for
6:    /* Main loop to build partitions. */
7:    /* Variable $m$ contains the number of partitions that access different variables. */
8:    /* Each element of array $P[]$ represents all the mutex bodies that access the same set of variables. */
9:    /* Each element of array $M[]$ contains a list of mutex bodies that access the same set of variables. */
10:   $m \leftarrow 0$
11:   $P[] \leftarrow \perp$
12:   $M[] \leftarrow \perp$
13:   foreach mutex body $B_i \in M_L$ do
14:     $found \leftarrow$ FALSE
15:     /* Check if $B_i$ accesses variables referenced by some existing partition. */
16:     /* Note that this loop will not execute for the first mutex body $B_i$. */
17:     for $j \leftarrow 1$ to $m$ do
18:       if $V_i \bigcap P[j] \neq \emptyset$ then
19:         $P[j] \leftarrow P[j] \bigcup V_i$
20:         add $B_i$ to mutex structure $M[j]$
21:         $found \leftarrow$ TRUE
22:       end if
23:     end for
24:     /* If $B_i$ does not have variables in common with other partitions then */
25:     /* create a new partition for it. */
26:     if $found =$ FALSE then
27:       $m \leftarrow m + 1$
28:       $P[m] = V_i$
29:       $M[m] = B_i$
30:     end if
31:   end for
32:   /* Replace original lock $L$ with $m$ locks; one for each partition. */
33:   /* This is only done if the previous analysis found more than one partition. */
34:   if $m > 1$ then
35:     for $j \leftarrow 1$ to $m$ do
36:       foreach mutex body $B(n, x)$ in mutex structure $M[j]$ do
37:         modify node $n$ to reference lock $L_j$
38:         modify node $x$ to reference lock $L_j$
39:       end for
40:     end for
41:   end if
42: end for
```

to perform whole program analysis. Function inlining and IPA information will help discover nested and ordered locking patterns for lock picking. This will also increase the chance of finding lock independent code.

The memory model also plays a crucial role. Notice that the analysis and optimization techniques developed for these memory semantics are also valid for weaker memory models. This is because the sequential memory model assumed in our work allows more memory interleavings than weaker memory models. Assuming the worst case scenario makes our analysis conservatively correct for other memory models. However, realizing that a weaker memory model is being used might increase the optimization opportunities. Consider for example a release consistent architecture [5]. Updates to shared memory variables are only observable at synchronization points. This allows the compiler to further eliminate conflict edges from the graph which will allow more optimization opportunities. We are currently incorporating release consistency semantics to the CSSAME model presented in this paper.

8.1 Implementation

The algorithms discussed in previous sections have been implemented[4] in a prototype compiler for the C language using the SUIF compiler system [8]. To avoid modifying SUIF's front-end we added support for `cobegin`/`coend` and `parloop` parallel structures via language macros. These macros re-define control structures of the C language so that the compiler can recognize them at the intermediate language level. The `cobegin`/`coend` structure is represented by a `switch` statement. A special index variable helps the compiler distinguish a regular `switch` statement from a `cobegin`. Each different `case` section will be executed by a different thread at runtime. Our system leverages on the SUIF runtime system to execute the parallel program. SUIF's runtime system is designed to run SPMD style programs. Our compiler annotates `cobegin` statements to be executed in parallel and modifies the index variable to be the thread id. Parallel loops are recognized using a similar technique. A `parloop` loop is a `for` loop with a special index variable. Since SUIF directly supports `parloop` style parallelism all the compiler has to do is mark selected `for` loops as parallel loops. Although our analysis techniques do not address parallel loops yet, users can write data parallel programs using our compiler. Currently, the compiler ignores `parloop` loops when applying optimizing transformations.

Once the program has been parsed by the SUIF front-end, the compiler creates the corresponding CCFG and its CSSAME form. Notice that we do not transform the input program to CSSA form. Instead we use FUD chains in the flow graph and display the source code annotated with the appropriate π and ϕ functions (Variables are not renamed

[4]The implementation is available at `http://www.cs.ualberta.ca/~diego/CSSAME/`

but referenced using line number information in the corresponding π or ϕ functions). The CCFG implementation is an extension of the sequential Control Flow Graph library provided by Machine SUIF [10]. The CCFG can be displayed using a variety of graph visualization systems. The flow graphs in this paper were generated with the GraphViz system [19]. The CSSAME form for the program can also be displayed as an option.

Besides optimization, mutual exclusion analysis can also be used to statically validate synchronization patterns in the program. For instance, the compiler may issue warning messages like unmatched `lock` and `unlock` operations or improperly nested locks. A limited form of data race detection capability is also built-in for inconsistent use of locks to protect shared variables. For instance, if modifications to a variable are not always protected by the same lock, the compiler will warn the user about a potential data race. A simple extension to Algorithm 4.5 allows the compiler to perform some semantic checking on the synchronization structure of the program. At the end of the algorithm, every `lock` or `unlock` node that is not part of a mutex body can be reported as a warning to the user. The following are some example warnings issued by our compiler when identifying mutex structures in the code:
(1) If there is a pair of nodes $n = \texttt{lock}(L_i)$ and $x = \texttt{unlock}(L_i)$ such that there is a control path from n to x then:

- If $n \in DOM(x)$ but $x \notin PDOM(n)$, then the lock may not be released on every execution.
- If $x \in PDOM(n)$ but $n \notin DOM(x)$, then the lock may not be held on every execution.

(2) Any remaining nodes in $p_i^{lock} \bigcup p_i^{unlock}$ are unmatched synchronization operations.
(3) After the CSSAME form is computed, any use inside a mutex body reached by conflict edges from nodes outside the mutex structure are references that are not protected by locks.

9 Conclusions and Future work

We have shown how the CSSAME form is unique in allowing new optimization opportunities by taking advantage of the semantics imposed by mutual exclusion synchronization. The reduction of memory conflicts across threads can improve the effectiveness of scalar optimization strategies. Furthermore, we have introduced new optimization techniques that are specifically targeted at explicitly parallel programs. We consider this a step forward in fully exploiting optimization opportunities in explicitly parallel programs. We plan to develop new optimization techniques to take advantage of the parallel and synchronization structure of these programs.

We are currently studying some multi-threaded applications to determine the effectiveness of the conflict reduction techniques. We expect to find more cases of protected uses than consecutive kills. We do not expect programmers to write code that deliberately makes consecutive killing definitions. Rather, these will likely result from previous optimization passes like constant propagation.

We are also investigating the representation of parallel loops in the CSSAME framework. With the inclusion of parallel loops we will incorporate barriers to the set of synchronization constructs recognized by CSSAME. Other research directions that we have planned include: study the effects of different memory consistency models on the CSSAME form (e.g., release consistency), study partial lock independence (akin to partial redundancy and common subexpressions), apply IPA information to propagate synchronization information and adapt other scalar optimizations using the CSSAME form.

Acknowledgements

The authors would like to thank Steve MacDonald for his suggestions and helpful comments during the preparation of this manuscript.

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] M. M. Brandis and H. Moessenboeck. Single-pass generation of static single-assignment form for structured languages. *ACM Transactions on Programming Languages and Systems*, 16(6):1684–1698, November 1994.

[4] D. Callahan, K. Kennedy, and J. Subhlok. Analysis of event synchronization in a parallel programming tool. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 21–30, Seattle, WA, March 1990.

[5] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[6] 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.

[7] D. Grunwald and H. Srinivasan. Data flow equations for explicitly parallel programs. *ACM SIGPLAN Notices*, 28(7):159–168, July 1993.

[8] 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.

[9] D. Harel. A linear-time algorithm for finding dominators in flow graphs and related problems. In *Symposium on Theory of Computing*, pages 185–194, May 1985.

[10] G. Holloway and C. Young. The Flow Analysis and Transformation Libraries of Machine SUIF. In *Proc. 2nd SUIF Compiler Workshop*, Stanford University, August 1997. URL: http://www.eecs.harvard.edu/hube.

[11] R. Johnson, D. Pearson, and K. Pingali. The program structure tree: Computing control regions in linear time. In *ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 171–185, Orlando, Florida, June 1994.

[12] 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.

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

[14] J. Lee, S. Midkiff, and D. A. Padua. Concurrent static single assignment form and concurrent sparse conditional constant propagation for explicitly parallel programs. Technical Report TR#1525, CSRD, University of Illinois at Urbana-Champaign, July 1997.

[15] 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.

[16] S. Masticola and B. Ryder. Non-concurrency analysis. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 129–138, San Diego, CA, May 1993.

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

[18] S. P. Midkiff and D. A. Padua. Issues in the optimization of parallel programs. In *1990 International Conference on Parallel Processing*, volume II, pages 105–113, St. Charles, Ill., August 1990.

[19] S. C. North and E. Koutsofios. Application of graph visualization. In *Proceedings of Graphics Interface '94*, pages 235–245, Banff, Alberta, Canada, May 1994. Canadian Information Processing Society. URL: http://www.research.att.com/-∼north/graphviz/.

[20] D. Shasha and M. Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

[21] V. C. Sreedhar and G. R. Gao. A linear time algorithm for placing ϕ-nodes. In *22nd Annual ACM Symposium on Principles of Programming Languages*, pages 62–73, New York, NY, USA, January 1995. ACM Press.

[22] H. Srinivasan, J. Hook, and M. Wolfe. Static single assignment for explicitly parallel programs. In *20th Annual ACM Symposium on Principles of Programming Languages*, pages 16–28, Charleston, S.C., January 1993.

[23] M. Wegman and K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

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