**University of Alberta**

**Library Release Form**

**Name of Author**: Diego Novillo

**Title of Thesis**: Analysis and Optimization of Explicitly Parallel Programs

**Degree**: Doctor of Philosophy

**Year this Degree Granted**: 2000



Diego Novillo
Av. Coronel Díaz 2164, 5 B
Buenos Aires, 1425
Argentina

**Date**: ____________________

**University of Alberta**

# Analysis and Optimization of Explicitly Parallel Programs

by

**Diego Novillo**

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

Department of Computing Science

Edmonton, Alberta
Spring 2000

**University of Alberta**

**Faculty of Graduate Studies and Research**

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Analysis and Optimization of Explicitly Parallel Programs** submitted by Diego Novillo in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

Jonathan Schaeffer
Supervisor

Ron Unrau
Co-Supervisor

Mariusz Klobukowski

Duane Szafron

Laurie Hendren
External Examiner

**Date**: ____________________

Para Lily.

Convertiste mi Sueño en Realidad.

# Abstract


In this thesis we introduce the CSSAME form, a new analysis framework for explicitly parallel programs that recognizes three fundamental elements of a parallel program: (1) parallel structure, (2) memory semantics, and (3) synchronization structure. By modeling these three elements in a single unified framework, a compiler can better exploit optimization opportunities in parallel programs.

We also develop a new synchronization analysis technique to detect mutual exclusion synchronization patterns that cannot be analyzed with existing techniques. We introduce the notion of multiple-entry/multiple-exit mutex regions and provide methods for validating mutual exclusion synchronization at compile-time. This analysis provides the basis for the elimination of superfluous memory conflict edges in the program's flowgraph, leading to a simpler representation and allowing more optimization opportunities.

We integrate reaching definition analysis and dead-code elimination into the CSSAME framework. Furthermore, we introduce new optimization techniques to reduce mutual exclusion synchronization overhead: Lock Picking, Lock Independent Code Motion and Mutex Body Localization. We study the effects of these transformations in the context of SPLASH and Java applications, prove their correctness, and provide algorithms that implement them.

# Acknowledgements

My first important discovery during this work was the realization that this is not an individual achievement. Far from it. Over the years many people have provided me with the necessary intellectual, spiritual and monetary support needed to cross the finish line. First and foremost, I want to thank my family: Lily, Nicky, Papá, Mamá, Mo, Enri y Ernie. Sin ustedes esto no hubiera sido posible. Gracias!

My long-time friend and wonderful educator Carlos Neetzel planted the first seeds of curiosity. Thank you Carlos for showing me what's beneath the covers.

My supervisors Dr Jonathan Schaeffer and Dr Ron Unrau provided superb guidance and support (both monetary and intellectual). Three different points of view are sometimes difficult to reconcile. But I soon learned the subtle art of steering out of trouble by letting them argue among themselves. With infinite patience they taught me the basics of research, compilers and parallel computing. Their heroic efforts converted my often convoluted writing and thought process into the organized document that you read today.

To my examining committee Dr Laurie Hendren, Dr Duane Szafron and Dr Mariusz Klobukowski: I thank you for your time and dedication in reading this document and providing valuable feedback. All the remaining errors and omissions are exclusively my fault. I am particularly grateful for the suggestions that Dr Hendren made to simplify the algorithms that analyze irregular mutex bodies in the code.

Special thanks to Dr Duane Szafron who started being one of my supervisors until I turned into the dark side of compilers, bits, bytes and pointers. Your high-level views of parallel computing helped educate an otherwise ignorant "metal-head". Rest assured that not all your efforts have gone to waste.

To my lab mates and friends Wally Lysz, Ivan Ourdev, Roel van der Goot, Steve MacDonald, Mark Brockington, Ian Parsons and David Woloschuk; thank you for the stimulating environment you helped create. Steve: Has anyone asked you what CSOC stands for? (E-mail `stevem@cs.ualberta.ca` for details).

I am also in debt with Ron Senda, my manager at the Research Support Group for the University of Alberta. Thank you for allowing me the time and resources to write and play with the compiler. I would also like to thank Jim Lemke, my current manager at Cygnus Solutions, for making it possible to finish writing the thesis during my initial time at Cygnus.

Finally, to my friends: Gracias muchachos!

Edmonton, March 2000

# Contents

# List of Tables

# List of Figures

# List of Definitions

# List of Algorithms

# List of Theorems and Lemmas

# Chapter 1

# Introduction

Parallel computers have the potential to solve complex problems much faster than conventional sequential computers. Unfortunately, the mere presence of multiple processors does not automatically guarantee better performance. Parallel programs must explicitly distribute the work among the available processors and coordinate their activities. In turn, this division of labor also affects the algorithm used to solve the problem. While some sequential algorithms lend themselves to parallel implementations, others do not.

Sequential algorithms amenable to parallelization have been extensively studied and existing tools can automatically turn some algorithms into their parallel counterpart. This approach, known as *implicit* or *automatic parallelization* works well on some application domains but it is not a universal solution (Blume and Eigenmann 1992; Eigenmann and Blume 1991). In this dissertation we are interested in algorithms that are parallel from the outset. These algorithms express the solution to a problem in terms of sub-problems to be solved concurrently. The necessary allocation of work to the different processes, coordination and data sharing are explicitly stated in the algorithm. Languages that support the implementation of explicitly parallel algorithms are called *explicitly parallel languages.*

In an explicitly parallel language, the programmer has full control over the parallelism in the program. This is an expressive model because it allows the user to take full advantage of the system capabilities. However, performance is still an issue; using an explicitly parallel language does not necessarily lead to optimum runtime performance. In addition to good algorithm design and implementation, an essential key to obtaining good performance is the compiler. The compiler is responsible for translating a program written in a high-level language to an equivalent program in a low-level language that the target architecture can understand. During this translation process the compiler applies optimizing transformations to the code to improve its performance. In general these transformations have an important property: they preserve the semantics of the original program (i.e., the optimized program behaves like the original one). In certain circumstances, however, optimizing transformations can alter the semantics of the program. Typical examples include transformations that trade-off floating point arithmetic precision in favor of speed.

To successfully transform a program the compiler must gather information about it. This process, known as *program analysis*, builds the necessary data structures representing the flow of control and data in the original program. This information is vital for the subsequent process of *program optimization* that improves the original program. It should be noted that the term *optimization* is really a misnomer. Optimizing transformations try to improve the original code but they make no guarantees that the transformation will actually be *optimal*. The transformations are intended to produce code that is *no worse* than the original one.

This thesis introduces novel compiler analysis and transformation techniques to optimize the performance of explicitly parallel programs. In the following sections we describe the problem in detail (Section 1.1), present our main contributions of this work (Section 1.2) and describe the organization of this thesis (Section 1.3).

## 1.1 The Problem

Arguably, the easiest way to develop a parallel program is to write sequential code and have the system automatically generate an equivalent parallel program. This process, known as *automatic* or *implicit parallelization*, has been the focus of intense research and development for over three decades.

Conceptually, this process works like any other optimizing transformation; the parallelizer (often built into the compiler) looks for constructs in the original program that can be executed concurrently without altering the original semantics. By executing multiple instructions simultaneously, the execution path of the program is shortened, thus reducing its runtime.

This approach to generating parallel code has been extremely successful in certain application domains. Traditionally, programs performing matrix and vector computations using regular loops are prime candidates for automatic parallelization. Many scientific problems in physics, engineering and chemistry fall into this category. Unfortunately, the state of the art in parallelizing technology has not advanced much beyond this. Parallelizing compilers are fundamentally limited by the need to preserve the original sequential semantics of the program. The transformations must be such that the resulting parallel program should produce exactly the same results as the sequential version. For many application domains implicitly parallelizing a sequential algorithm is seldom better than using an explicitly parallel algorithm from the outset. For instance, the parallel version of the well-known *quicksort* algorithm, a very good sequential algorithm, performs very poorly compared to PSRS, an explicitly parallel sorting algorithm (Shi and Schaeffer 1992).

The recognition of these limitations has resulted in an increased demand for explicitly parallel languages. An explicitly parallel language provides language constructs or library functions that allow the programmer to describe concurrent activity inside the program. This added flexibility is a double-edged sword; programmers are free to specify parallel algorithms any way they choose at the potential expense of increased programming complexity. For some time now, researchers have developed new programming models, programming environments and automatic validation techniques to simplify the development of parallel programs. However, developing parallel programs is complex in another dimension: performance. Most of the existing work in the language area has addressed expressibility and flexibility issues. Programming environments like Enterprise (Schaeffer et al. 1993) provide an integral framework for developing parallel programs based on common parallel constructs. Analysis tools exist to statically detect deadlock patterns

(Masticola and Ryder 1993) and shared memory conflicts (Emrath et al. 1992; Helmbold and McDowell 1994; Callahan et al. 1990). New languages and programming models are being constantly introduced; each typically well-suited to a few specific classes of problems. However, these developments rarely address performance, which is, in our view, the main reason for using a parallel computer in the first place.

Little research has been done on making compilers understand explicitly parallel code for the purpose of optimization. Typically, existing systems and tools rely on the programmer to develop efficient code. The system understands explicitly parallel semantics only to the extent of mapping the program to the target architecture. Little or no attempt is made to optimize the code. In fact, current commercial compilers treat explicitly parallel sections of the code as a "black box" and leave them untouched. There is a good reason for this limitation: transformation techniques for optimizing sequential programs cannot be directly applied to explicitly parallel code because they may generate incorrect transformations (Midkiff and Padua 1990). The techniques developed in this thesis fill part of the void. We present a unified framework for analyzing and optimizing explicitly parallel programs. The optimizations described here fall into two classes: the adaptation of sequential optimizations to a parallel environment; and the direct optimization of the parallel and synchronization structure of the program.

## 1.2 Summary of Major Contributions

The techniques developed in this thesis can be organized into two categories: analysis and transformation techniques. Analysis techniques allow the compiler to reason about an explicitly parallel program. We prove correctness properties about the analysis and provide algorithms that implement the techniques. Transformation techniques use the information gathered by the analysis and convert parts of the program into a more efficient but semantically equivalent form. We prove correctness properties about the transformations and provide algorithms that implement them. We have also implemented most of these algorithms in the SUIF compiler infrastructure (Hall et al. 1996).

We apply them to several explicitly parallel programs and show that these optimizations can result in significant improvements in performance. The following sections provide an overview of the specific contributions of this work.

### 1.2.1 Analysis Techniques

#### Static Single Assignment Form for Parallel Programs

This thesis introduces the Concurrent Static Single Assignment form with Mutual Exclusion (CSSAME). CSSAME[1] is an intermediate program representation based on the the well-known Static Single Assignment (SSA) form (Cytron et al. 1991). The SSA form is based on the fundamental premise that every memory variable in the intermediate program can only be assigned once. If a program is transformed to comply with this condition we say that the program is in SSA form.

An SSA form for parallel programs with interleaving memory semantics must take into account that write and read operations to a given variable can take place simultaneously from different processes. The CSSAME form extends the single assignment concept to the parallel case. It is based on the Concurrent Static Single Assignment (CSSA) form (Lee et al. 1997b). CSSAME extends the CSSA form to support two important synchronization mechanisms, namely mutual exclusion and barrier synchronization. Chapter 4 presents a formal description of the CSSAME framework.

#### Mutual Exclusion Synchronization Detection

Mutual exclusion synchronization is used when a process needs to have exclusive access to a shared resource. Exclusive access to a shared resource prevents simultaneous modifications which might lead to an inconsistent state. We will model mutual exclusion using `lock` and `unlock` operations. Exclusive access to a shared resource is requested using a `lock` operation. Once the requesting thread is done accessing the resource, it calls `unlock` to free the resource and allow other threads to access it. All the instructions executed

[1]Pronounced *sesame*.

between the `lock` and the corresponding `unlock` operation are said to be inside a *mutual exclusion section*. Other names for mutual exclusion section include *mutex body* and *critical section*. In the context of concurrent programs, mutual exclusion is typically used to access shared variables that might be otherwise modified by several concurrent threads.

Since synchronization operations can occur in arbitrary sections of the code, the mutual exclusion sections defined by `lock` and `unlock` operations can be difficult to discern. In this thesis we develop a new analysis technique to detect mutual exclusion sections in the program. Although techniques exist to detect mutual exclusion sections, they are limited in the types of locking patterns that they can detect. We formulate a different algorithm for detecting critical sections that can cope with irregular locking patterns in the code. This analysis provides the foundation for all the transformations that optimize the synchronization structure of the program, and can also be used to warn the programmer about illegal locking patterns.

### 1.2.2 Optimizations

We apply the CSSAME analysis framework to perform two types of optimizations: (1) the adaptation of known sequential transformations to the parallel case and (2) the development of new transformations that target the parallel and synchronization structure of the program directly.

Current research efforts in the field are geared towards the first type of transformations (Knoop et al. 1996; Lee et al. 1998; Lee et al. 1999). In this thesis we adapt a sequential dead-code elimination algorithm to the parallel case.

Transforming the parallel and synchronization structure of explicitly parallel code has received less attention (Krishnamurthy and Yelick 1996; Novillo et al. 1998). We contribute new algorithms to eliminate synchronization overhead from explicitly parallel programs: lock picking, lock-independent code motion and mutex body localization.

### Dead-Code Elimination

When a statement computes a value that is not used anywhere else in the program we say that that computation is *dead*. Dead code is usually removed from the program because it serves no useful purpose. In this thesis we adapt a sequential dead-code elimination algorithm (Cytron et al. 1991) to the parallel case.

### Lock Picking

Using lock information collected during the construction of the CSSAME form, it is possible to detect `lock` and `unlock` operations that are not needed in the program. As a simple case, consider a sequential program or a sequential section of a parallel program. Since there is no parallel activity, any synchronization operation in that section is not necessary and can be removed. We call this transformation *lock picking*.

### Lock-Independent Code Motion (LICM)

Mutual exclusion can become a performance bottleneck if used excessively because it restricts parallel activity in the program. In general it is desirable to reduce the size and number of mutual exclusion sections in the code. Lock-Independent Code Motion (LICM) tries to reduce the size of mutual exclusion sections by moving code outside mutual exclusion sections. This technique scans all the mutual exclusion regions in the program looking for interior code that does not need to be protected by the corresponding lock. The algorithm can move expressions, statements and even whole control structures out of critical sections.

### Mutex Body Localization (MBL)

Mutex Body Localization is a new transformation that converts references to shared memory into references to local memory inside critical sections of the code. This transformation can potentially create more lock-independent code that can be later optimized by LICM.

## 1.3 Thesis Organization

The rest of this thesis is organized as follows:

- Chapter 2 provides background information and related work about parallel programming, synchronization models and optimizing compilers. It also provides details about the necessity of adapting sequential optimization techniques to work on explicitly parallel programs. The specific language model that we assume in the rest of this thesis is introduced: an explicitly parallel language with interleaving memory semantics and three different synchronization mechanisms (mutual exclusion, barriers and event variables).

- Chapters 3 and 4 describe the analysis framework that we use to reason about parallel programs. We describe the Concurrent Control Flow Graph (CCFG) that represents the control and synchronization structure of parallel programs, the technique used to identify mutual exclusion synchronization patterns and the CSSAME form.

- Chapter 5 builds on the CSSAME form to develop the following optimizing transformations: concurrent dead-code elimination, lock-independent code motion, mutex body localization, lock picking and lock partitioning.

- Experimental results are presented in Chapter 6. We illustrate the benefits of using the CSSAME framework and the effects of the different transformations on selected parallel programs taken from SPLASH (Singh et al. 1992) and TreadMarks (Keleher et al. 1994). We also investigated the potential benefits of our optimizations on programs written in Java. We found that the generic nature of Java's thread-safe libraries leads to correct but conservative implementations that are often overly synchronized. When our optimizations are applied to sample Java programs we observed up to a factor of 4 improvement in runtime compared to the original parallel program. In fact, because the same libraries are used for sequential programs, we were able to get between 10% and 25% improvement in *sequential* programs when our optimizations are applied.

- Conclusions and future work are the subject of Chapter 7.

## 1.4 Summary

With low-cost multiprocessor systems now being ubiquitous, the need for tools to maximize parallel performance has never been greater. This thesis represents a significant step forward in improving the capabilities of compilers for parallel programs. In particular, we expect these techniques to have a significant impact in high-level concurrent or thread-based languages. Of particular importance in these environments is the ability of the compiler to understand synchronization operations which can be a source of substantial overhead in some applications.

# Chapter 2

# Background

This chapter introduces the fundamental concepts used as the foundation for the techniques developed in this thesis. The discussion starts with an overview of the more popular parallel programming models, including the specification of parallel activity, memory semantics and synchronization constructs (Section 2.1).

The discussion continues with a description of the structure and responsibilities of a typical optimizing compiler. The emphasis is on the data structures and program representations used in the optimization phase of the compilation process (Section 2.2).

Finally, Sections 2.3, 2.4 and 2.5 provide background information about the field of analysis and optimization of explicitly parallel programs. Techniques used in sequential compilers cannot be directly applied to parallel programs. We will describe the reasons for this limitation and survey existing work in the area. This discussion will motivate the new techniques developed in the rest of this dissertation.

## 2.1 Parallel Programming Models

Several issues must be considered in a parallel programming environment: specification of parallel activity (language model), data sharing semantics (memory model) and synchronization operations to order the access to shared resources (synchronization model).

**Language model**. The specification of parallel activity determines how the different processes participate in a computation. There are two types of parallelism: *task* and *data*. In a *task-parallel* program, different threads execute different sections of the program on different data elements. Conversely, in a *data-parallel* program, different threads execute the same code on different data elements.

**Memory model**. Unlike sequential programs, the different processes that execute a parallel program do not necessarily have access to the same memory address space. The memory can be shared among the processes, or distributed. The choice of memory model will have a significant impact on the implementation and even on the algorithms used.

**Synchronization model**. Synchronization is necessary to protect the integrity of resources shared by several processes. It prevents a process from computing with stale or incomplete data.

### 2.1.1 Language Model

For a long time, research in the field of parallel compilation has focused on the automatic transformation of sequential programs into their parallel equivalent (Gupta and Banerjee 1992; Wilson et al. 1994). The compiler analyzes the program looking for sections of the code that can be executed in parallel without affecting the original data dependencies in the program.

Parallelizing compilers are very useful for some application domains. They typically excel in numeric and scientific applications involving computations on regular data structures like matrices. Unfortunately, there are some important problem domains that parallelizing compilers cannot handle efficiently (Blume and Eigenmann 1992; Eigenmann and Blume 1991) (e.g., sorting, searching, sparse matrix computations, etc). These shortcomings are not always due to limitations in the parallelization techniques used. For some applications, the best sequential algorithms contain data and control dependencies that current automatic parallelization techniques cannot handle. To overcome these limitations, parallelizing compilers provide a set of annotations and directives so that the programmer can direct the actions of the parallelizer. Even these

```
/* Start N threads to execute different
 * sections of code concurrently.
 */
cobegin
   T1: begin
      statements
    end

   T2: begin
      statements
    end
   ...
   TN: begin
      statements
    end
coend
```

(a) A task-parallel program.

```
/* Start N threads to execute the same
 * code concurrently. Each thread executes
 * with a different value of i.
 */
parloop (i, 1, N) {
   stmt1;
   stmt2;
   ...
   stmtM;
}
```

(b) A data-parallel program.

Figure 2.1: Syntax for specifying parallel activity in a program.

extensions are often not enough; often the best solution is to solve the problem using a parallel algorithm from the outset (Shi and Schaeffer 1992). All the techniques and algorithms developed in this thesis work on explicitly parallel programs. Our goal is not to extract parallelism from a sequential program but to analyze and optimize a program that is already parallel. This applies to programs that are explicitly parallel from the outset and to the output of an automatic parallelization tool.

We assume that explicitly parallel programs start as a single thread of computation. New threads are logically created when execution reaches a parallel section in the program. 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. There are a variety of mechanisms for expressing parallel activity. Some examples include `cobegin/coend` constructs, `fork` statements and parallel loops.

We will represent task-parallel programs using `cobegin/coend` constructs (Figure 2.1(a)) and data-parallel programs using parallel loops (Figure 2.1(b)). The program fragments in Figure 2.1 launch N threads that execute independently and join with the invoking thread at the end of the parallel section. The threads created by the `cobegin/coend` construct will execute different code sections while the threads created by the `parloop` loop will

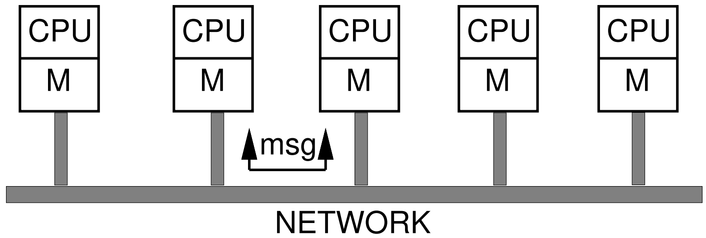



Figure 2.2: A distributed-memory system. Processors have their own memory.

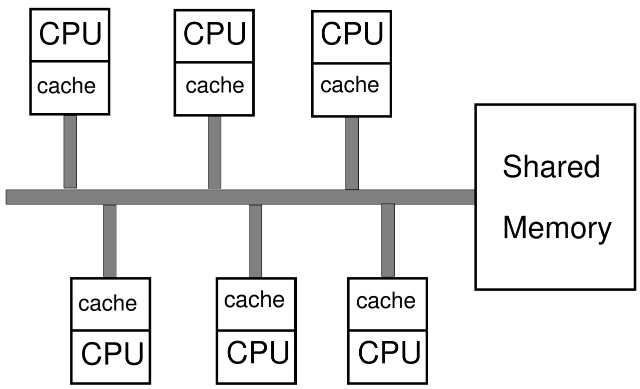



Figure 2.3: A shared-memory system. Processors share the same address space.

execute the same piece of code. With these two constructs it is possible to express both task-parallel and data-parallel algorithms.

### 2.1.2 Memory Model

Memory can be shared or distributed among the processors in the system. On a distributed-memory system, each processor has its own local memory which cannot be accessed by other processors in the system (Figure 2.2). Interprocessor communication is based on message passing. Data is sent from one processor to another via data communication primitives *send* and *receive*.

In contrast to the distributed approach, a shared-memory system provides a single address space that can be accessed by all the processors in the system (Figure 2.3). Traditionally, shared memory has been provided in hardware with processors connected to a common memory pool through a shared bus. These systems, known as Symmetric Multiprocessors (or SMPs), suffer from scalability problems; beyond a certain number the performance of

SMP systems degrades greatly because of the increased traffic on the shared memory bus.

To address the scalability problem, research has focused on providing a shared memory image on top of physically distributed hardware. These systems, known as Distributed Shared Memory (or DSM) or Non-Uniform Memory Access systems (NUMA), mask the distributed nature of the memory by providing an abstraction that transforms shared memory references into messages between different memory modules.

A sometimes heated debate exists in the parallelism community about the relative benefits of shared-memory versus distributed-memory systems. Supporters of the shared memory model argue that its unified data access notation makes for simpler and easier to maintain programs. Any communication required to access the common memory is transparently handled by the system. The current trend is for these two types of architectures to merge into hybrid architectures with features from both types of systems.

While this is a convenient programming model, the overhead of repeated shared-memory references can restrict the performance of the program significantly. The focus of current research into shared-memory systems is in minimizing communication due to shared-memory traffic. This has produced compiler techniques, caching algorithms and latency-hiding techniques at the hardware and operating system level. In this work we assume that threads run in a shared address space with interleaving semantics (i.e., updates to shared memory made by one thread are immediately visible to the other threads). Programs share memory via shared variables.

### 2.1.3 Synchronization Model

The analysis techniques discussed in this document rely on the effects that synchronization operations have on the flow of data in the parallel program. The algorithms developed in this thesis support three standard synchronization constructs, namely mutual exclusion, events and barriers:

- Mutual exclusion is used to serialize references to shared variables in the program. We will assume that programmers use standard `lock` and `unlock` instructions to serialize access to shared variables. Both instructions operate on *lock variables* which can only be referenced in a `lock` or `unlock` statement. Furthermore, we assume that `lock(L)` reads and writes to the lock variable $L$ and `unlock(L)` only writes to $L$.

`lock(L)` blocks the calling thread until it is granted exclusive access to the lock variable `L`. If a thread $t_2$ tries to acquire a lock already held by another thread $t_1$, $t_2$ will block until $t_1$ releases the lock. If multiple threads try to acquire the lock simultaneously, exactly one is guaranteed to succeed. The other threads are forced to wait.

`unlock(L)` releases the lock on `L` and allows one of the threads waiting on the lock to proceed.

- Event synchronization is supported using event variables. An event variable is an integer with two possible values, *posted* and *cleared*. Three operations apply to an event variable $e$:

  `set(e)` sets event variable $e$ to *posted*.

  `wait(e)` if $e$ is set to *cleared*, it blocks the calling thread until $e$ is set to *posted*.

  `clear(e)` sets $e$ to *cleared*.

  Event synchronization is used as a signaling mechanism between threads. By using events, the programmer can introduce a partial order in the execution of concurrent threads. Assume that some computation $B$ in thread $T_2$ can only execute after thread $T_1$ has produced another computation $A$. This relation can be implemented by using an event variable $e$ that is `set` by $T_1$ immediately after computing $A$ and `waited` by $T_2$ immediately prior to computing $B$. Our work does not address event synchronization directly; all the support for event synchronization is derived from the precedence algorithms in (Lee et al. 1997a).

- Barriers are used in algorithms that need to proceed in phases. A `barrier(b, N)` instruction forces the calling thread to wait until `N` threads have executed the statement `barrier(b, N)`.


Figure 2.4: A high-level view of the compilation process.

## 2.2 Optimizing Compilers

A compiler analyzes an input program written in one language (source code) and transforms it into a semantically equivalent program in another language (object code). During translation an optimizing compiler applies certain transformations to the input program to improve its efficiency. There are two fundamental ways of measuring efficiency: performance and space. Most optimizing transformations are meant to improve performance. In certain situations, space considerations are more important (e.g., systems with limited amounts of memory and/or registers).

We should point out that the transformations applied by an optimizing compiler are generally not optimal; they merely *attempt* to improve certain aspects of the program. Optimizing transformations try to be as aggressive as possible without modifying the original semantics of the program. To achieve this the optimization algorithms always err on the safe side; a transformation will only be applied if it is valid for every possible execution of the program. To summarize, an optimizing transformation must be aggressive but conservatively correct.

This section starts with an overview of a typical compiler system. Compilers have two major components: the front-end, which is responsible for recognizing and validating the input program; and the back-end, which translates the input program into the target language and applies optimizing transformations to make the program more efficient (Figure 2.4). Special attention is given to the back-end of the compiler; we will only briefly describe the compiler front-end (an in-depth description of this topic can be found in (Aho et al. 1986)).

### 2.2.1 Front-End

Before the program can be optimized and translated into code for the target machine, the compiler must understand its lexical and syntactic structure. The front-end of the compiler converts the string of characters representing the input program into data structures that convey all the information needed by the back-end to transform the program and generate object code. The recognition of the input program is done in three phases, namely lexical analysis, syntax analysis and intermediate code generation (Figure 2.5).

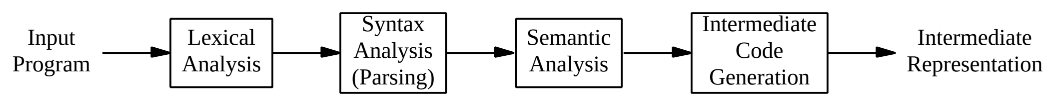



Figure 2.5: The front-end analyzes and prepares the program for optimization.

#### Lexical Analysis

This phase reads the stream of characters that make up the input program and groups them into *tokens*. Tokens are symbols with a predetermined meaning in the grammar of the input language (i.e., the *words* of the language). This *tokenization* process produces a more synthetic version of the input program that simplifies the task of subsequent phases. For example, given the following stream of characters representing an assignment statement

```
foo = bar + 30.4 - foo
```

a lexical analyzer might produce the following seven tokens

```
IDENT  ASSIGN  IDENT  PLUS  NUM   MINUS  IDENT
 foo     =      bar    +    30.4    -     foo
```

Limited error checking is performed at this phase. Basically, the lexical analyzer can only determine whether a string of characters is a valid token of the input language. The hierarchical grouping of tokens into statements is performed by the syntax analyzer.

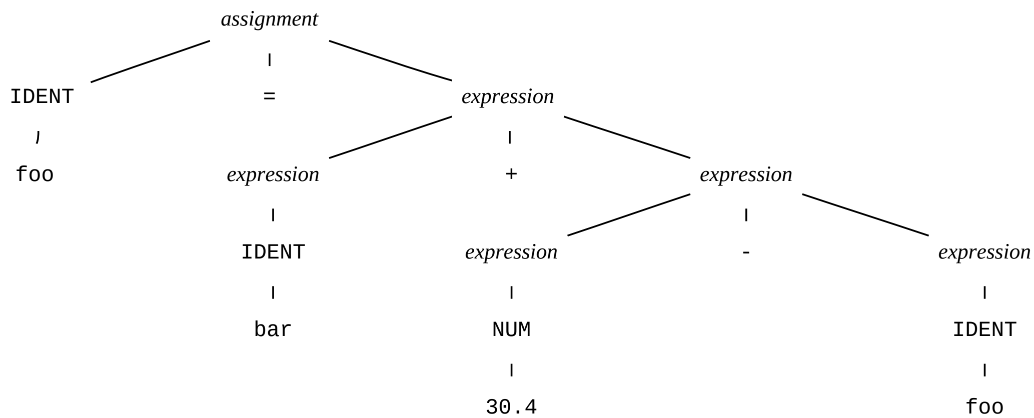



Figure 2.6: Parse tree for the statement `foo = bar + 30.4 - foo`.

### Syntax and Semantic Analysis

The syntax analyzer, also known as parser, uses the grammar rules of the input language to group the tokens into statements. Statements are hierarchical groupings often represented by *parse trees*. Information contained in parse trees is used to validate the syntax of the input program and generate intermediate code used for optimization and final object code generation.

Figure 2.6 shows the parse tree corresponding to the statement `foo = bar + 30.4 - foo`. Interior nodes of the tree correspond to grammar constructs (e.g., statements, expressions, declarations, etc); leaves correspond to the individual tokens recognized by the lexical analyzer.

Grammar rules are defined recursively in terms of statements, expressions, procedures and control structures. Semantic analysis is also performed during this phase. It mainly involves checking expressions to detect operations that are not allowed by the typing rules of the language (e.g., multiplying a string by a floating point number).

### Intermediate Code Generation

Once the program syntax has been verified, the compiler generates intermediate code which is a more synthetic representation of the original program. The intermediate representation used by the compiler often resembles assembly language for an abstract machine. By separating the language (front-end) from the architecture (back-end), it is possible to re-use the same optimization and code generation techniques for a variety of input languages. Furthermore, the simpler form of this intermediate language simplifies the task of optimizing and generating object code. Returning to our running example, the expression `foo = bar + 30.4 - foo` is translated to the following intermediate form in SUIF (Stanford University Intermediate Form) (Hall et al. 1996):

```
1: ldc  nd#4 = 3.04e+01    /* Load nd#4 with constant 30.4 */
2: add  nd#3 = .bar, nd#4  /* Add nd#3 = bar + nd#4 */
3: sub  .foo = nd#3, .foo  /* Subtract foo = nd#3 - foo */
```

In this code fragment, the symbols `nd#i` are temporary variables used internally by the compiler and actual program variable names are preceded by a ".". All the analysis and transformation techniques performed by the compiler are applied to this intermediate representation. The amount of detail provided by the intermediate representation depends on the type of optimization being performed. Optimizing compilers typically have more than one intermediate representation, each suited for different transformations. For example, high-level transformations like loop transformations are typically performed by the front-end while low-level transformations like code scheduling are typically done by the back-end (code scheduling reorders the generated instructions to take advantage of the target processor).

### 2.2.2 Back-End

The compiler back-end is responsible for applying optimizing transformations to the intermediate code and generating the object code that will execute on the real machine. The front-end for compilers for both sequential and parallel languages use similar methodologies. The techniques for recognizing and validating the input program are well-known and do not vary much when moving from the sequential to the parallel case. However, fundamental changes are necessary to the compiler's back-end when moving from the sequential to the parallel case.

There are also significant differences between compiler techniques for explicitly parallel languages (like the ones developed in this thesis) and the techniques used in parallelizing compilers. Parallelizing compilers analyze sequential programs to generate parallel code with sequential semantics. On the other hand, compilers for explicitly parallel languages analyze and optimize programs that already have parallel semantics.

### Optimizing Transformations

The compiler front-end acquires very little knowledge of what the program actually does. Optimization is possible when the compiler understands the flow of control in the program (control-flow analysis) and how the data is transformed as the program executes (data-flow analysis). Both types of analysis are discussed in Sections 2.4 and 2.5.

Analysis of the control and data-flow of the program allows the compiler to improve the runtime performance of the code. Many different optimizations are possible once the compiler understands the control and data-flow of the program. The following are a few of the more popular optimization techniques used in standard optimizing compilers:

**Algebraic simplifications**. Expressions are simplified using algebraic properties of their operators and operands. For instance, $i + 1 - i$ is converted to 1. Other properties like associativity, commutativity and distributivity are also used to simplify expressions.

**Constant folding**. Expressions for which all operators are constant can be evaluated at compile time and replaced with their value. For instance, the expression $a = 4 + 3 - 8$ can be replaced with $a = -1$. This optimization (usually performed by the front-end) yields best results when combined with constant propagation (page 23).

**Redundancy elimination**. There are several techniques that deal with the elimination of redundant computations. Some of the more common ones include:

*Loop-invariant code motion.* Computations inside loops that produce the same result for every iteration are moved outside the loop.

*Common sub-expression elimination.* If an expression is computed more than once on a specific execution path and its operands are never modified, the repeated computations are replaced with the result computed in the first one.

*Partial redundancy elimination.* A computation is partially redundant if some execution path computes the expression more than once. This optimization adds and removes computations from execution paths to minimize the number of redundant computations in the program. It encompasses the effects of loop-invariant code motion and common sub-expression elimination.

**Register allocation**. Registers are memory locations inside the processor itself that are extremely fast and scarce. Register allocation tries to keep memory traffic within the CPU registers as much as possible.

### Code Generation

Final target code consists of machine or assembly code for the target architecture. Further optimizations are enabled during this translation. Register allocation and code scheduling are typically applied during this phase. Code scheduling refers to a family of instruction re-ordering techniques that take advantage of specific features of the processor (e.g., pipelining, VLIW, super-scalar features, etc).

## 2.3 Analysis and Optimization of Explicitly Parallel Programs

In 1990 Midkiff and Padua published a study that showed how optimizing transformations designed for sequential programs may fail when applied to explicitly parallel code (Midkiff and Padua 1990). The core of the problem is that techniques for sequential languages have no concept of concurrent activity, they assume a single thread of execution. Consequently, they cannot assert whether it is safe to apply the transformations.

Current work-arounds to this problem involve disabling optimizations in parallel sections of the program and/or restricting data sharing between threads. Both are inappropriate because they are too restrictive. This means that the compiler can only optimize the sequential parts of the program. The compiler should "understand" parallel code and be able to make valid optimizing transformations. A classic example of how sequential compilers fail on explicitly parallel code is shown in Figure 2.7. The program shows two threads sharing a common array. Thread $T_0$ (the *producer*) creates new values while thread $T_1$ (the *consumer*) waits for $T_0$ to generate all the values before doing its work. The two threads are synchronized using a busy-wait loop on variable *done*. When thread $T_0$ finishes updating the array, it sets variable *done* to 1 which terminates the `while` loop in thread $T_1$.

A common transformation used in optimizing compilers is called *constant propagation*. Basically, a constant propagation algorithm replaces variables by their values if they are known to be constant. Consider variable *done*; since a sequential constant propagation analyzer does not know about the parallel structure of the program, it will produce incorrect transformations. If the compiler considers that $T_0$ and $T_1$ execute in sequence, it will conclude that variable *done* is always 1 when control reaches the `while` loop in $T_1$. Therefore, constant propagation will effectively remove the busy-wait loop and the program will likely produce the wrong results at runtime.

This example illustrates the fundamental reason why we need compilers to understand explicitly parallel code. Concurrent threads of activity on shared data introduce data dependencies that a sequential compiler cannot see because it assumes a single thread of execution.

There are other elements in a parallel program that a compiler must understand, namely the synchronization and memory models. Different synchronization schemes will impose different constraints on how data is shared. As we will see in later sections this can create more opportunities for the compiler to apply more aggressive optimizations.

```
done = 0;
cobegin
  T0: begin
    for (i = 0; i < N; i++)
        A[i] = produce(i);
    done = 1;
  end

  T1: begin
    while (done == 0)
      ; /* busy-wait */
    for (i = 0; i < N; i++)
        print(A[i]);
  end
coend
```

(a) Original program.

```
done = 0;
cobegin
  T0: begin
    for (i = 0; i < N; i++)
        A[i] = produce(i);
    done = 1;
  end

  T1: begin
    while (1 == 0)  /* Always false! */
      ; /* busy-wait never executed */
    for (i = 0; i < N; i++)
        print(A[i]);
  end
coend
```

(b) Constant propagation eliminates synchronization.

Figure 2.7: Constant propagation problems in an explicitly parallel program.

## 2.4 Control-Flow Analysis

The goal of control-flow analysis is to discover the control structure of the program. This task might seem trivial when one examines the original source code, but recall that the compiler does not deal with the original code. Depending on the intermediate representation used, when the code is converted to its intermediate form, all the high-level control constructs like loops and conditionals are sometimes lost. Even if the control information was preserved, programmers can still write obfuscated code that hide the high-level control structures of the program.

The control-flow of the program is often represented in a graphical form called the *control-flow graph*. The nodes of the graph, called *basic blocks*, represent a non-branching sequence of statements (i.e., execution starts with the first instruction in the group and it only leaves the block after the last instruction has been executed). The edges of the graph represent possible execution paths in the flow of control (i.e., conditionals, loops, etc.).

### 2.4.1 The Control-Flow Graph

The control-flow graph (also known as the *flowgraph*) is a graphical representation of the control structure of the program. Its nodes represent computations and its edges represent the flow of control. The nodes of a flowgraph are called *basic blocks*.

**Definition 2.1 (Basic block)** A *basic block* is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without any possibility of branching except at the end (Aho et al. 1986).

□

Formally, a control-flow graph is defined as a directed graph $G = \langle N, E, \mathit{begin}, \mathit{end}\rangle$ such that $N$ is the set of basic blocks (or nodes), $E \subseteq N \times N$ is the set of control-flow edges, *begin* is the unique entry point to the graph and *end* is the unique exit point from the graph. An edge between basic blocks $n$ and $m$ is denoted $n \rightarrow m$. We say that node $n$ is the *immediate predecessor* of $m$ and node $m$ is the *immediate successor* of $n$. Similarly we define the sets of $\mathit{Succ}(n)$ and $\mathit{Pred}(n)$ to be the sets of immediate successors and predecessors of $n$ respectively.

```
a = f();
b = g();
c = h();

if (a + b < c) {
  d = c;
} else {
  d = a + b;
  c = a * b;
}
```

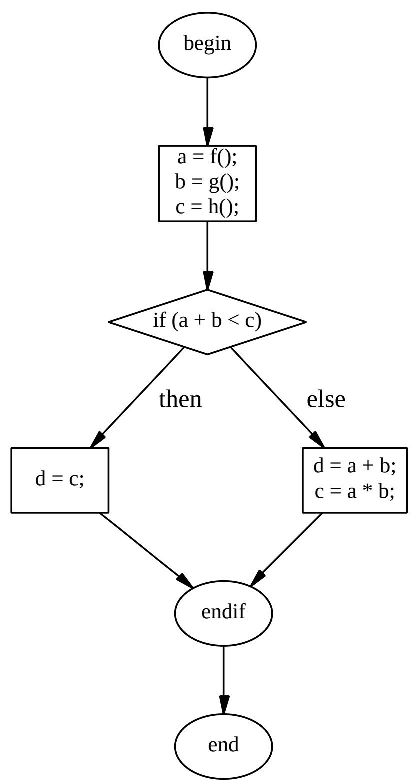



Figure 2.8: A sequential program and its control-flow graph.

Figure 2.8 shows a sample flowgraph for a sequential program. While there is little variation in the conventions used to represent flowgraphs for sequential programs, there does not exist a unique notation to represent flowgraphs for parallel programs. The different representations share commonalities, but some include extra edges to represent synchronization and have different notions of basic blocks.

### Parallel Flow Graph

Srinivasan and Grunwald introduce the *Parallel Flow Graph* (PFG) (Grunwald and Srinivasan 1993). In their language model synchronization is specified using `Post` and `Wait` statements and parallel sections in the code are specified using `cobegin/coend` or `parallel_sections/end_parallel_sections`.

The nodes of a PFG represent extended basic blocks. An *extended basic block* is a basic block with at most one `Wait` statement at the start of the block and at most one `Post` statement at the end of the block. Statements demarking parallel sections are denoted by `cobegin` and `coend` nodes in the graph. There are three types of edges: a *sequential control-flow edge* represents sequential flow of control within sequential parts of the program. A *parallel control-flow edge* represents parallel control flow. It connects a `cobegin` node with its immediate successors and a `coend` node with its immediate predecessors. A *synchronization edge* is a directed edge between a node containing a `Post` statement to a node containing the corresponding `Wait` statement.

### Extended Flow Graph

Srinivasan, Hook and Wolfe introduce the *Extended Flow Graph* (EFG) (Srinivasan et al. 1993). Parallel activity is specified using `Parallel Sections`. Each section within a `Parallel Sections` construct has its own identifying name. The only synchronization supported is the `Wait(sec)` clause which can only be used at the beginning of a section. The `Wait(sec)` command causes the invoking section to wait until section `sec` has finished.

The EFG is composed of two separate abstractions; the *Parallel Control Flow Graph* (PCFG) which represents the sequential sections of the code and the *Parallel Precedence Graph* which represents the parallel sections.

The PCFG is a standard control-flow graph with one special node called *supernode* that represents an entire `Parallel Sections` construct. Each section within a `Parallel Sections` is a node of a *Parallel Precedence Graph*. Synchronization between parallel sections is represented with directed edges between the corresponding nodes in the PPG. In turn, each node of the PPG is expanded into a PCFG representing the code inside the section.

### Concurrent Control Flow Graph

Lee, Midkiff and Padua introduce the *Concurrent Control Flow Graph* (CCFG) (Lee et al. 1997b). It is similar to the Parallel Flow Graph but since the memory model that they use allows concurrent modifications to shared memory locations, the CCFG also contains conflict edges between basic blocks that contain conflicting memory references (i.e., at least one of the basic blocks is attempting to modify that location).

The nodes of a CCFG are called *concurrent basic blocks* and are exactly like the *extended basic blocks* of a PFG. The flowgraph representation used in this thesis is based on the CCFG. We will describe CCFGs in detail in Chapter 3.

## 2.4.2 Common Graph Concepts

In this section we define several relations between nodes in a control-flow graph that are commonly used by the analysis algorithms. In what follows we assume a control-flow graph $G = \langle N, E, Entry_G, Exit_G \rangle$ and two nodes $x, y \in G$.

**Definition 2.2 (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 $y \in DOM^{-1}(x)$. Note that every node always dominates itself. □

**Definition 2.3 (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 2.4 (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 $x \in PDOM^{-1}(y)$. Note that every node always post-dominates itself. □

**Definition 2.5 (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 $x \in SPDOM^{-1}(y)$. □

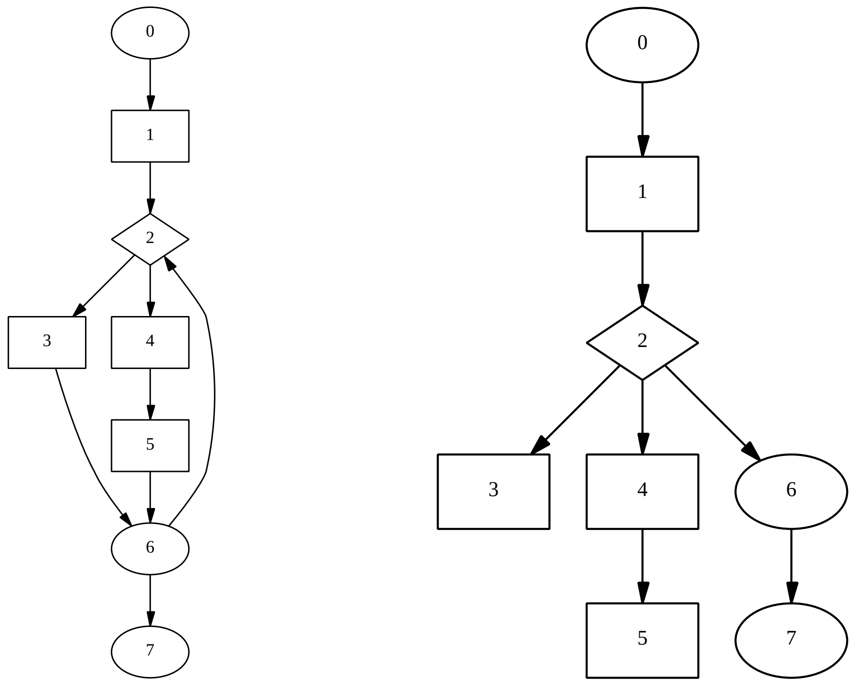
**Definition 2.6 (Dominance frontier)** The *dominance frontier* for node $x$, denoted $DF(x)$ is the set of all nodes $y$ in the flowgraph such that $x$ dominates an immediate predecessor of $y$ but it does not dominate $y$. □

**Definition 2.7 (Immediate dominator)** If $x\ DOM\ y$, we say that node $x$ is the *immediate dominator* of node $y$, denoted $x\ IDOM\ y$, if $x$ is the last dominator of $y$ on any path from the entry node to $y$. □

**Definition 2.8 (Dominator tree)** The *dominator tree* is defined recursively using the dominance relation between the nodes in the graph. The root of the dominator tree is the entry node to the graph. The children of a node $n$ in the dominator tree are the nodes immediately dominated by $n$ in the flowgraph.

□

We illustrate these concepts using the flowgraph shown in Figure 2.9(a). The entry node (node 0) dominates every node in the graph. Consequently its dominance frontier is empty. Nodes $1, 2, 6$ and $7$ post-dominate node 0 because every path $0 \rightarrow 7$ must go through those nodes. The dominance frontier for node 4 is node 6 because node 4 dominates an immediate predecessor of node 6 (i.e., node 5), but it does not dominate node 6 itself (i.e., there is a path from 0 to 6 that does not include node 4). Using the dominance relation on the nodes of the graph we obtain the dominance tree shown in Figure 2.9(b). The tables in Figures 2.10 and 2.11 show the dominance and post-dominance relations for the nodes in the example flowgraph.




(a) Flowgraph. (b) Dominator tree.

Figure 2.9: An example flowgraph and its dominator tree.

## 2.5 Data-Flow Analysis

A data-flow analyzer explores all the possible executions of the program to determine how it transforms the data it manipulates. A fundamental property of data-flow analysis is that it must guarantee that the information it gathers is valid for *every* possible execution of the program. Otherwise, decisions based on this analysis could yield erroneous results.

This section describes some of the more common data-flow analyses found in optimizing compilers. Two popular data-flow analysis frameworks are discussed: iterative data-flow analysis and the Static Single Assignment form. We also survey proposed analysis techniques for explicitly parallel languages based on these data-flow frameworks.

| Node (n) | $DOM(n)$ | $DOM^{-1}(n)$ | $DF(n)$ |
|---|---|---|---|
| 0 | $\{0\}$ | $\{0,1,2,3,4,5,6,7\}$ | $\emptyset$ |
| 1 | $\{0,1\}$ | $\{1,2,3,4,5,6,7\}$ | $\emptyset$ |
| 2 | $\{0,1,2\}$ | $\{2,3,4,5,6,7\}$ | $\emptyset$ |
| 3 | $\{0,1,2,3\}$ | $\{3\}$ | $\{6\}$ |
| 4 | $\{0,1,2,4\}$ | $\{4,5\}$ | $\{6\}$ |
| 5 | $\{0,1,2,4,5\}$ | $\{5\}$ | $\{6\}$ |
| 6 | $\{0,1,2,6\}$ | $\{6\}$ | $\emptyset$ |
| 7 | $\{0,1,2,6,7\}$ | $\{7\}$ | $\emptyset$ |

Figure 2.10: Dominance sets and dominance frontiers for Figure 2.9.

| Node (n) | $PDOM(n)$ | $PDOM^{-1}(n)$ |
|---|---|---|
| 0 | $\{0,1,2,6,7\}$ | $\{0\}$ |
| 1 | $\{1,2,6,7\}$ | $\{0,1\}$ |
| 2 | $\{2,6,7\}$ | $\{0,1,2\}$ |
| 3 | $\{3,6,7\}$ | $\{3\}$ |
| 4 | $\{4,5,6,7\}$ | $\{4\}$ |
| 5 | $\{5,6,7\}$ | $\{4,5\}$ |
| 6 | $\{6,7\}$ | $\{0,1,2,3,4,5,6\}$ |
| 7 | $\{7\}$ | $\{0,1,2,3,4,5,6,7\}$ |

Figure 2.11: Post-dominance sets for the flowgraph in Figure 2.9.

### 2.5.1 Common Data-Flow Problems

Data-flow problems model properties about various program objects at specific points in the program. The information gathered when solving a specific problem is then used by the optimizer to make the actual transformations.

#### Reaching Definitions

A variable $v$ is *defined* (denoted $D_v$) every time a new value is assigned to it. We say that a definition $D_v$ of $v$ *reaches* a certain point $p$ in the program if there exists a path $r$ between $D_v$ and $p$ such that $r$ contains no definitions to $v$. For example, the program in Figure 2.12 contains three definitions of variable $a$, namely $D_a^1$ at line 1, $D_a^2$ at line 4 and $D_a^3$ at line 7. Reaching definition analysis on this program should determine that definition $D_a^1$ reaches the use of $a$ at lines 2, 4 and 6 but it does not reach line 8 because of definition $D_a^3$ at line 7.

```
a = 4;              /* D_a^1 */
b = a + 3;          /* U_a^1 */
if (b > 10) {
   a = b * 2;       /* D_a^2 */
}
print a;            /* U_a^2 */
a = a + 10;         /* D_a^3, U_a^3 */
print a;            /* U_a^4 */
```

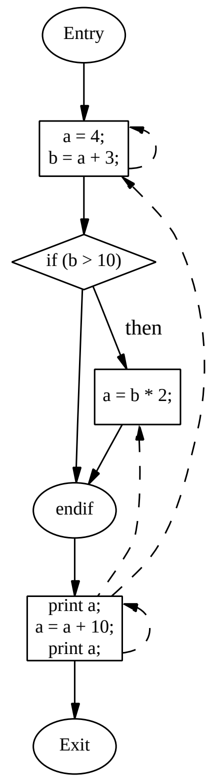



Figure 2.12: Example of the reaching definitions problem.

| Def | *reached-uses* |
|---|---|
| $D_a^1$ | $\{U_a^1, U_a^2, U_a^3\}$ |
| $D_a^2$ | $\{U_a^2, U_a^3\}$ |
| $D_a^3$ | $\{U_a^4\}$ |

(a) Reached uses for each definition of $a$.

| Use | *reaching-defs* |
|---|---|
| $U_a^1$ | $\{D_a^1\}$ |
| $U_a^2$ | $\{D_a^1, D_a^2\}$ |
| $U_a^3$ | $\{D_a^1, D_a^2\}$ |
| $U_a^4$ | $\{D_a^3\}$ |

(b) Reaching definitions for each use of $a$.

Figure 2.13: Reaching definitions and reached uses sets for the program in Figure 2.12.

**Definition 2.9 (Use-def chains)** Reaching definition information is usually stored in *use-def chains* or *ud-chains* which are lists of definitions reaching a particular use of a variable. □

Use-def chains for variable $a$ are shown as dashed arrows in the control-flow graph for the program (Figure 2.12). Other data structures of interest include *reached-uses* and *reaching-defs* sets which are defined as follows:

**Definition 2.10 (Reached-uses set)** Given a definition $D_v$ for variable $v$, the set *reached-uses* for $D_v$ is the set of all uses of $v$ that are reached by $D_v$. □

**Definition 2.11 (Reaching-defs)** Given a use $U_v$ of variable $v$, the set *reaching-defs* for $U_v$ is the set of all definitions for $v$ that can reach $U_v$. □

Note that in collecting reaching definition information for this program we have said that definition $D_a^1$ reaches line 6. This might appear counter-intuitive because there appears to be another definition in the path from line 1 to line 6, namely definition $D_a^2$ at line 4. However, definition at line 4 is not always executed therefore the conservatively correct decision is to assume that *both* definitions, $D_a^1$ and $D_a^2$, reach line 6. Reaching definitions and reached uses sets for variable $a$ are shown in Figure 2.13.

### Live Variables

A variable $v$ is *live* at a certain point $p$ in the program if the value of $v$ at $p$ could be used along some path starting at $p$. Otherwise, we say that $v$ is *dead* at $p$. Going back to the example program in Figure 2.12, the value of $b$ computed at line 2 is live at line 3 but it becomes dead at line 5 because it is not used anymore.

### Available Expressions

An expression $a + b$ is *available* at a point $p$ in the program if all the paths from the entry node to point $p$ in the graph compute $a + b$. The notion of availability is used in optimizations like redundancy elimination. If an expression is repeatedly computed without its operands being modified, then redundant computations can be removed.

### 2.5.2 Iterative Data-Flow Analysis

Iterative data-flow analysis is the traditional method for solving data-flow problems. Data-flow information is collected in sets that represent the information needed by each particular problem. Traditionally, optimizing transformations are phrased in terms of data-flow problems. For instance, in the case of constant propagation each element of the data-flow set corresponds to a different variable in the program.

The analysis is performed by setting up and solving systems of equations, known as *data-flow equations*, that describe the local effects that each basic block has on the data-flow sets. The propagation of data-flow properties is done locally to each basic block and the results are aggregated over all the basic blocks to determine global properties of the program. Each data-flow problem must define appropriate data-flow sets and equations needed to gather the required information.

Data-flow information is typically stored in four main sets: *in* is the set representing information entering the block, *out* is the information that exits the block, *kill* is the information invalidated (or killed) by the block and *gen* is the information generated locally by the block. In general, the equations are set up so that they follow the natural flow of control of the program. In other words, the set *out* is defined in terms of *in*, *gen* and *kill*. These are known as *forward* data-flow problems. But for some other problems, known as *backward* data-flow problems, the data-flow equations and their associated iterations proceed backwards.

Once set up, data-flow equations are solved iteratively from an initial set of values. The most common implementation of iterative data-flow analyzers uses bit-vectors to represent the sets in the data-flow equations. This is why this is sometimes called *bit-vector analysis*. More information about these techniques can be found in (Aho et al. 1986) and (Muchnick 1997).

#### Iterative Data-Flow Analysis for Explicitly Parallel Programs

Grunwald and Srinivasan developed data-flow equations to compute reaching definition information on explicitly parallel programs with `cobegin/coend` parallel sections (Grunwald and Srinivasan 1993). They assume a weak memory consistency model in which parallel sections are required to be data independent; memory updates are done at specific points in the program using `copy-in/copy-out` semantics. They support event-based synchronization synchronization using `set` and `wait` operations.

Knoop, Steffen and Vollmer developed a bit-vector analysis framework for parallel programs with interleaving memory semantics (Knoop et al. 1996). They show how to adapt standard optimization algorithms to their framework. However, they do not incorporate synchronization operations in their analysis. They use this framework to adapt *lazy code motion* optimization which is a redundancy elimination method.

### 2.5.3 Static Single Assignment Form

Static Single Assignment (SSA) is a relatively new intermediate representation that is becoming increasingly popular because it leads to efficient algorithmic implementations of data-flow analyzers and optimizing transformations (Cytron et al. 1991). The SSA form is based on the premise that program variables are only assigned once. Multiple assignments to the same variable create new versions of the variable. In essence, the SSA form makes all the use-def chains explicit in the program, because every use of a variable is reached by exactly one definition.

Actual programs are seldom in SSA form initially because variables tend to be assigned multiple times; not just once. An SSA-based compiler modifies the program representation so that every time a variable is assigned in the code, a new version of the variable is created. Different versions of the same variable are distinguished by subscripting the variable name with its version number. Variables used in the right-hand side of expressions are renamed so that their version number matches that of the most recent assignment. Notice that it is not always possible to statically determine what is the most recent assignment for a given use. These ambiguities are the result of branches and loops in the program flow of control. To solve this ambiguity, the SSA form introduces the so-called $\phi$ functions. $\phi$ functions merge multiple incoming assignments to generate a new definition; they are placed at points in the program where the

```
1: a = 4
2: b = a + 3
3: if (a > 3) {
4:     print a
5:     a = a + 3
6: }
7:
8: b = 5
9: print a + b
```

(a) Original program.

```
1: a_1 = 4
2: b_1 = a_1 + 3
3: if (a_1 > 3) {
4:     print a_1
5:     a_2 = a_1 + 3
6: }
7: a_3 = φ(a_1, a_2)
8: b_2 = 5
9: print a_3 + b_2
```

(b) Program in SSA form.

Figure 2.14: An example sequential program and its SSA form.

flow of control causes more than one assignment to be available (essentially, a $\phi$ functions are needed at dominance frontier nodes).

Figure 2.14 shows a sequential program and its corresponding SSA form (Figures 2.14(a) and 2.14(b) respectively). Notice that every assignment in the program introduces a new version number for the corresponding variable. Every time a variable is used, its name is replaced with the version corresponding to the most recent assignment for the variable. Now consider the use of variable $a$ in line 9. There are two assignments to $a$ that could reach line 9; the assignment at line 1 and the assignment inside the `if` statement at line 5. To solve this ambiguity, SSA introduces a $\phi$ function for $a$ which merges both assignments to create a new version of $a$ ($a_3$). The semantics of the $\phi$ function dictate that $a_3$ will take the value from one of the function's arguments. The specific argument returned by the $\phi$ function is not known until runtime.

### Static Single Assignment for Explicitly Parallel Programs

Srinivasan, Hook and Wolfe developed a Static Single Assignment (SSA) framework for explicitly parallel programs (Srinivasan et al. 1993). Their analysis framework works on the `Parallel Sections` model (page 26). Two different merge operators are used; $\phi$ and $\psi$ functions. A $\phi$ function serves the same purpose as in sequential programs, it is placed at nodes that represent merge points in the program. $\psi$ functions model multiple parallel updates; they are placed at synchronization points in the program if two or more concurrent sections modify the same variable.

Lee, Midkiff and Padua propose a Concurrent SSA framework (CSSA) for explicitly parallel programs and interleaving memory semantics (Lee et al. 1997b). Our work builds on the CSSA form; a more detailed description can be found in Chapter 4. Lee *et al.* also adapt some sequential optimizing transformations to the parallel case using CSSA (Lee et al. 1998; Lee et al. 1999).

### 2.5.4 Other Approaches to Optimizing 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 (Shasha and Snir 1988).

Krishnamurthy and Yelick extended cycle detection analysis to incorporate additional information from synchronization in the program (Krishnamurthy and Yelick 1996). 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.

Recent research efforts in the area have focused on the Java language. Since Java is a multi-threaded language, its class libraries must support concurrent accesses by multiple threads of execution. This is supported at the language level using *synchronized methods*, also known as *monitors*, which are a variation of the traditional mutual exclusion section. An important aspect of optimizing Java programs is reducing the overhead imposed by the thread-safe nature of Java's libraries. Diniz, Rinard and Whaley have developed several techniques to reduce the impact of synchronization in Java programs (Whaley and Rinard 1999; Diniz and Rinard 1998).

## 2.6 Summary

Modern compilers are organized around two major phases: *analysis* and *synthesis*. During analysis, the compiler extracts detailed information about the program. In particular the analysis phase discovers how the program is structured and how it manipulates its data. The optimization phase uses this information to transform the original program into an equivalent but more efficient version. In this context, efficiency is usually associated with *performance*; we want to produce code that executes as fast as possible on the target architecture. Finally, the synthesis phase generates object code that can be executed on the target machine.

While analysis and optimization techniques for sequential languages are well-known, these techniques cannot be used in explicitly parallel programs that share memory. Concurrent execution, data sharing and synchronization operations affect the control and data flow of the program in ways that the sequential techniques are unable to handle. There have been recent advances in developing analysis frameworks for explicitly parallel programs and adapting traditional optimization techniques such as constant propagation and dead-code elimination to the parallel case. However, there has been less emphasis on optimizing the parallel and synchronization structure of the program itself.

In the following chapters we introduce novel analysis techniques that incorporate both the parallel and synchronization structure of the program into a unified framework for analyzing and optimizing explicitly parallel programs.

# Chapter 3

# Analyzing Explicitly Parallel Programs

In an explicitly parallel program with shared memory semantics, the use of a shared variable $v$ can be reached by any definition of $v$ in another concurrent thread. However, synchronization constructs may prevent some variable definitions from being visible to other threads. For example, consider the program in Figure 3.1. If the compiler ignores the mutual exclusion regions created by the lock operations, it 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 $a = 3$.

This chapter introduces the foundations for the analysis framework developed in Chapter 4. We start with a description of the Concurrent Control Flow Graph (CCFG) (Section 3.1). Section 3.2 describes the process used to build the CCFG for a given program. We then use the CCFG to analyze the synchronization patterns in the program to gather non-concurrency information. As observed in Figure 3.1, synchronization can reduce data dependencies across concurrent threads in the program. This reduction of data dependencies may allow more aggressive optimization in subsequent transformation passes. In this work we support three types of synchronization operations: events, mutual exclusion and barriers (Section 3.3).

```
cobegin                 /* Begin concurrent execution */
  T0: begin             /* Launch thread T0 */
    if (b > 0) {
      b = 3 / a;
    }
    lock(L);
    a = a + b;
    unlock(L);
  end

  T1: begin             /* Launch thread T1 */
    f(a);
    lock(L);
    a = 3;              /* This kills the assignment to a in T0 */
    b = b + g(a);       /* Variable a is always 3 */
    unlock(L);
  end
coend
```

Figure 3.1: Mutual exclusion can reduce data dependencies across threads in a parallel program.

## 3.1 Concurrent Control Flow Graph

A Concurrent Control Flow Graph (CCFG) (Lee et al. 1997b) is similar to its sequential counterpart, the Control Flow Graph (Aho et al. 1986). It represents the control structure of a parallel program including the parallel constructs `cobegin/coend` and `parloop`. In addition, a CCFG contains edges to represent memory conflicts across concurrent threads and event synchronization. We extend the CCFG so that each `lock`, `unlock` and `barrier` operation is represented by a separate node.

**Definition 3.1 (Variable references)** Variables are *referenced* every time their values are read or modified by the program. Read references are also known as *uses*, while write references are also known as *definitions*. □

**Definition 3.2 (Shared variable reference conflicts)** Two variable references in different threads *conflict* if (a) both reference the same variable, (b) one of them is a write reference, and, (c) the threads can execute concurrently. □

**Definition 3.3 (Concurrent basic block)** A *concurrent basic block* is a basic block (Aho et al. 1986) 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.

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

3. Synchronization operations `lock`, `unlock` and `barrier` are placed in their own block.

4. Parallel control instructions `cobegin`, `coend` and `parloop` are placed in their own block. □

**Definition 3.4 (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 3.5 (Concurrent Control Flow Graph (CCFG))**
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.

3. $E = E_f \bigcup E_s \bigcup E_c$ is the set of edges in the graph such that:

   (a) $E_f$ is the set of control flow edges. These edges have the same meaning as in a sequential Control Flow Graph.

   (b) $E_s$ is the set of edges representing event synchronization. These are directed edges that join related `set` and `wait` nodes in concurrent threads.

   (c) $E_c$ is the set of conflict edges. Conflict edges are bi-directional edges that join any two concurrent basic blocks that conflict. There is a label on a conflict edge that represents the memory operations done at each end of the edge. There are two kinds of conflicts:

      i. *def-use*: one of the nodes writes to the shared variable and the other one reads from it. These conflicts are labeled `DU(v)`, where $v$ is the name of the variable being accessed.

ii. *def-def*: both nodes write to the shared variable. These conflicts are labeled `DD(v)`, where $v$ is the name of the variable being modified. □

**Definition 3.6 (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 3.7 (Control path)** Given two nodes $x$ and $y$ in a CCFG $G$, a path from $x$ to $y$ is a *control path* if it only contains edges in $E_f$. □

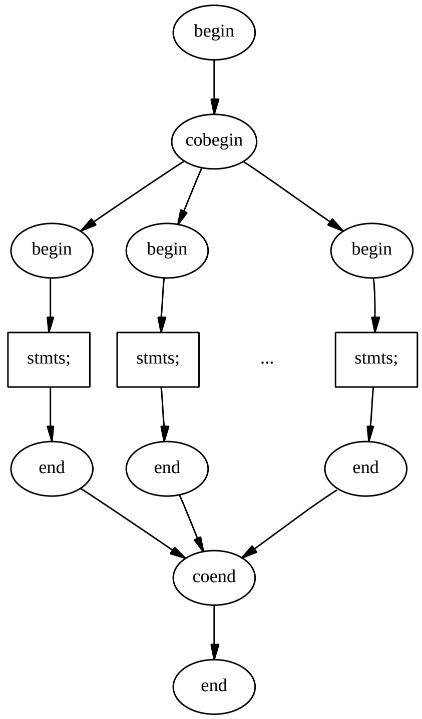
### 3.1.1 Graphical Representation of a CCFG

This section describes the graphical notation we use to represent CCFGs. Figures 3.2(a) and 3.2(b) show the representation for `cobegin/coend` and `parloop` constructs respectively. Figure 3.2(c) illustrate the representation of event synchronization edges.

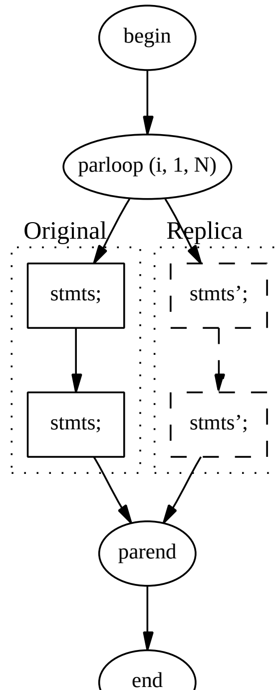
Graph nodes are represented using three different shapes. Ellipses represent entry and exit nodes for the graph, loops, parallel structures (`cobegin/coend` and `parloop`) and nested scopes in the source program. Header nodes for conditional statements are represented using diamonds. Finally, rectangles represent concurrent basic blocks. Control flow edges are represented using solid lines. Conflict edges are represented with dotted lines. Dashed lines represent event synchronization edges.

Each `cobegin` node has one outgoing control edge for each child thread it launches. Graphically, each thread is represented as a sub-graph rooted at the `cobegin` node (Figure 3.2(a)). All the children threads join at the `coend` node. Conflict edges always join nodes in threads that share at least one common `cobegin` node.

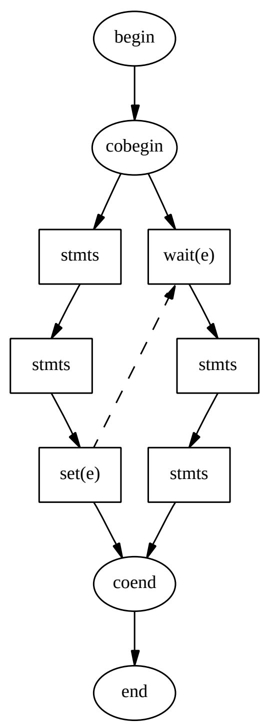
We experimented with two different ways of representing parallel loops. Since a parallel loop is not really an iterative control structure, we initially represented parallel loops as a `cobegin/coend` with one thread. Each node inside the `parloop` structure had the property of being concurrent with itself. Therefore, the algorithms and data structures have to support self-referencing




(a) `cobegin`/`coend` construct.




(b) `parloop` construct.




(c) Event synchronization edges.

Figure 3.2: Representation of parallel constructs and synchronization in a CCFG.

conflict edges. This is particularly important in building the CSSAME form for the program (Chapter 4).

Although this representation was enough for our purposes, it can be confusing to visualize and it does not permit certain analyses used in the literature (like cycle detection (Shasha and Snir 1988)). The other method to represent parallel loops is to replicate the body of the loop and consider it like a `cobegin/coend` structure with two threads: the **original** and the **replica** (Figure 3.2(b)). This representation is identical to the `cobegin/coend` representation, conflict edges join distinct nodes (there are no self-referencing conflicts) and it facilitates the design of some of the analysis algorithms proposed in the literature (Krishnamurthy and Yelick 1996; Lee et al. 1999). From an implementation point of view, this representation has the drawback of potentially doubling the memory requirements. In subsequent sections we use this representation to simplify the explanation of some algorithms. However, in our current implementation we do not create replicas of parallel loop bodies.

Event synchronization operations (`set` and `wait`) are represented in the flowgraph using directed edges from `set` nodes to the corresponding `wait` node. Notice that `set` and `wait` are the only synchronization operations that create additional edges in the CCFG. This is used during synchronization analysis to compute guaranteed precedence ordering (Section 3.3.3). Mutual exclusion and barrier synchronization are supported but no additional edges are required by the synchronization analysis phase. An example of an explicitly parallel program and its CCFG are illustrated in Figures 3.3 and 3.4.

## 3.2 Building the CCFG

Algorithm 3.1 builds the concurrent control flow graph for an explicitly parallel program $P$. It consists of three phases: (a) placement of nodes and control edges, (b) placement of conflict edges and (c) placement of synchronization edges.

Graph nodes and control edges are created using a slightly modified version of a standard algorithm to build control flow graphs (Aho et al. 1986). The modification allows the original algorithm to recognize the `cobegin/coend`

```
a = 0;
b = 0;
cobegin
  T0: begin
    lock(L);
    a = 5;
    b = a + 3;
    if (b > 4) {
      a = a + b;
    }
    x = a;
    unlock(L);
  end

  T1: begin
    lock(L);
    a = b + 6;
    y = a;
    unlock(L);
  end
coend
print(x, y);
```

Figure 3.3: A task parallel program.

and `parloop` constructs. Basic blocks are built using a linear scan of all the statements in the program. This step builds basic blocks, not concurrent basic blocks. Subsequent phases of the algorithm will split the basic blocks to create concurrent basic blocks, and incorporate conflict and synchronization edges to the base graph.

**Algorithm 3.1** 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 (Aho et al. 1986).
2: Add conflict edges (Algorithm 3.3).
3: Add synchronization edges (Algorithm 3.4).

Once the basic structure of the flowgraph has been built, conflict and synchronization edges are added to the graph. To add conflict edges, the graph is traversed looking for nodes that can execute concurrently and access the same memory location in a conflicting manner. Algorithm 3.2 is used to determine whether two arbitrary nodes in the graph can execute concurrently. The algorithm assumes the existence of two data structures:

$Thread(n)$ is the thread that contains node $n$. Threads are assumed to have

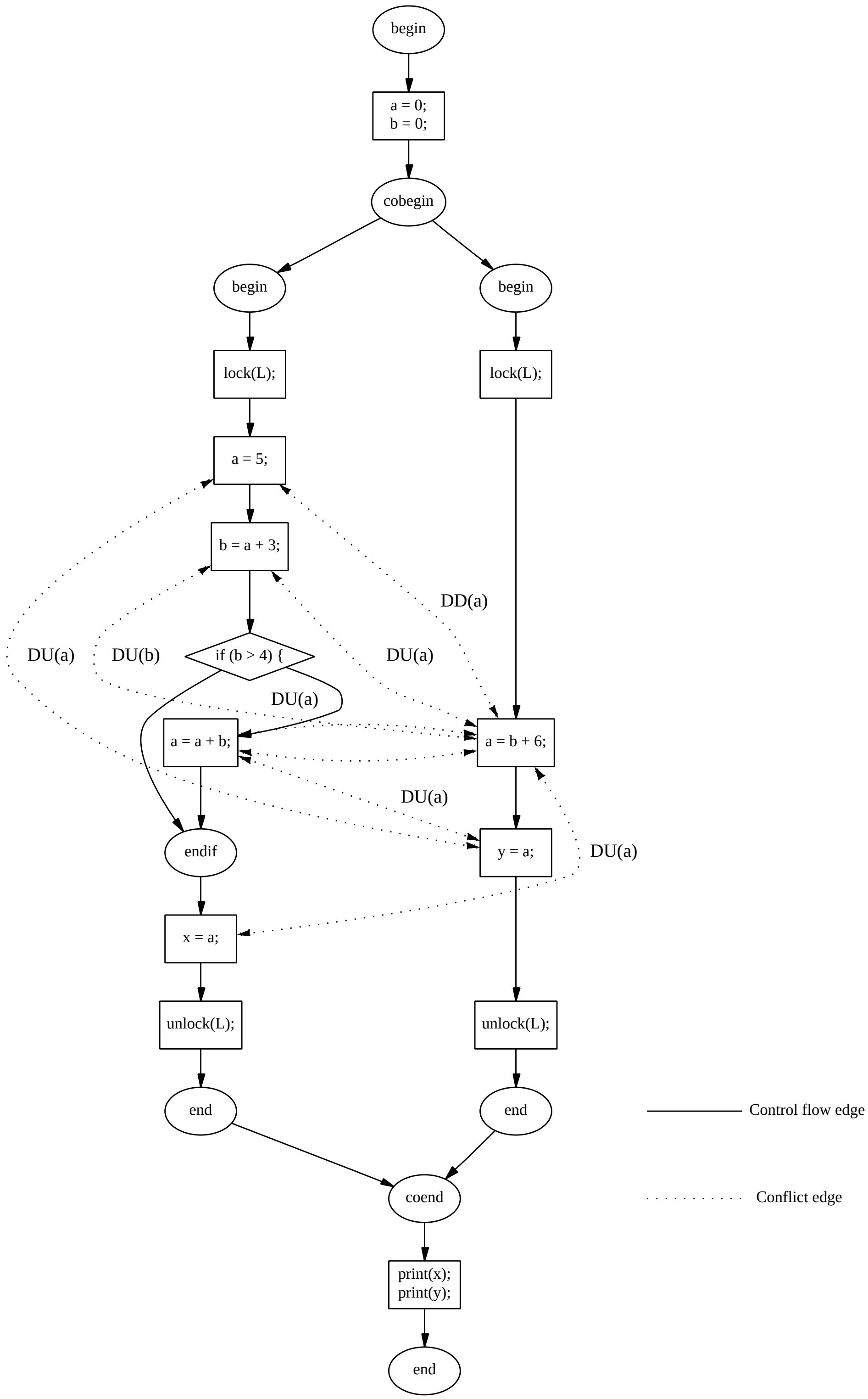



Figure 3.4: Concurrent Control Flow Graph for the program in Figure 3.3.

a unique id computed automatically by the compiler. The sequential parts of the program are always executed by thread $T_{seq}$.

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

**Algorithm 3.2** 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.

```
function conc(a, b)
/* If a or b are in a sequential region, they cannot be concurrent. */
if Thread(a) = T_seq ∨ Thread(b) = T_seq then
   return FALSE
end if

/* If a and b have a common parloop node in their ParAncestors set, they are concurrent. */
if ∃n ∈ ParAncestors(a) s.t. n = parloop ∧ n ∈ ParAncestors(b) then
   return TRUE
end if

/* If a and b have a common cobegin node in their */
/* ParAncestors set and they are on different threads */
/* and they are not the same node, then they are concurrent. */
if ∃n ∈ ParAncestors(a) s.t. n = cobegin ∧Thread(a) ≠ Thread(b) ∧ a ≠ b then
   return TRUE
end if

/* None of the previous tests succeeded. The nodes are not concurrent. */
return FALSE
```

Concurrent nodes with memory conflicts are marked as conflicting and split up to create concurrent basic blocks according to the rules given in Definition 3.3. Conflict edges are created to join the conflicting nodes (Algorithm 3.3). Notice that at this stage we do not use the non-concurrency information that can be gathered from the synchronization structures of the program. As we will discuss in Section 3.3, it is generally more convenient for synchronization analysis to have the basic CCFG already built. In practice, however, this analysis could be performed in conjunction with synchronization analysis.

When implementing the compiler, we discovered that it is easier to build concurrent basic blocks from the outset than it is to build maximal basic blocks and then split them up. The main reason is that when splitting basic blocks one must take care of boundary conditions so that no empty basic blocks are created. What we implemented is a two pass algorithm that will first scan the program and determine conflict lists at the level of instructions. During the concurrent basic block building pass, the conflict list in each instruction is checked to see if the instruction should be added to the current block or a new block be created. This is more memory intensive, but it simplified our implementation. For clarity of presentation we have decided to describe them as two separate phases.

**Algorithm 3.3** Add conflict edges.

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 $E_c$ added.

```
E_c ← ∅
foreach a ∈ N do
  foreach b ∈ N do
    /* Call Algorithm 3.2 (conc) to determine whether a and b are concurrent */
    if (conc(a, b) = TRUE) ∧ (a conflicts with b) then
      E_c ← E_c ⋃{(a, b)}
    end if
  end for
end for
foreach (a, b) ∈ E_c do
  Split blocks a and b to comply with definition 3.3.
end for
```

The last step in the construction of the CCFG is to add directed synchronization edges for related `set` and `wait` operations in the program (Algorithm 3.4). For every pair of nodes `set` and `wait` the algorithm checks if they can execute concurrently and operate on the same synchronization variable. If so, a directed edge from the `set` node to the `wait` node is added.

**Algorithm 3.4** Add synchronization edges.

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 $E_s$ added.

```
E_s ← ∅
/* For every event variable v add an edge from each set(v) to every wait(v). */
foreach a ∈ N do
  foreach b ∈ N do
    if conc(a, b) = TRUE then
      if (a = set(v)) ∧ (b = wait(v)) then
        E_s ← E_s ⋃ {(a, b)}
      end if
    end if
  end for
end for
```

## 3.3 Synchronization Analysis

Parallel programs use synchronization to order the access to shared data by the different threads in the program. Typically, synchronization operations introduce non-concurrency among otherwise concurrent regions of the program. The goal of synchronization analysis is to determine which nodes in concurrent sections of the program will not execute concurrently. This information is used to disregard memory conflicts from the CCFG that cannot occur at runtime due to synchronization restrictions. Reducing the number of memory conflicts gives more freedom to the compiler when applying optimizing transformations. Furthermore, information about synchronization semantics allows the development of techniques to validate the synchronization structure of the program.

In this work we support three types of synchronization: mutual exclusion, events and barriers. Section 3.3.1 develops new techniques to analyze mutual exclusion synchronization patterns in parallel programs. Techniques for statically validating mutual exclusion are discussed in Section 3.3.2. We use existing synchronization analysis techniques to gather non-concurrency information for `set/wait` and `barrier` operations (Jeremiassen and Eggers 1994; Lee et al. 1997b) (Sections 3.3.3 and 3.3.4).

### 3.3.1 Mutex Synchronization

Given an arbitrary statement $s$ in a program and a lock variable $L$, a mutex structure analyzer should be able to answer the question "does $s$ execute under the protection of lock $L$?". The answer to that question should be one of *always*, *never* or *sometimes*.

In the context of this work, the answers *never* and *sometimes* are equivalent. If the compiler cannot assert that statement $s$ will always be protected by $L$ at runtime then the conservatively correct decision is to assume that $s$ is never protected by $L$. Furthermore, if the analysis determines that $s$ is sometimes protected and sometimes not, this information could be used to warn the user about an anomalous locking pattern.

### Motivation

Existing work on mutual exclusion synchronization is based on a structural definition of mutex bodies (Krishnamurthy and Yelick 1996; Masticola and Ryder 1993; Novillo et al. 1998). A mutex body is indicated by a pair of `lock` and `unlock` nodes. All the graph nodes dominated by the `lock` node and post-dominated by the `unlock` node are part of the mutex body. Although correct, this notion of mutex body fails to identify some valid locking patterns present in some programs (i.e., the mutex body recognizer responds *never* too often).

Initially, we had only considered traditional single-entry, single-exit mutex bodies (Novillo et al. 1998) but we soon discovered that some programs contain mutex bodies that do not fit that structure. For instance, consider the code fragment in Figure 3.5. This routine is part of a quicksort algorithm taken from the sample application programs bundled with the TreadMarks DSM system (Keleher et al. 1994). This routine grabs a piece of work to be done from a shared stack. We are interested in the mutual exclusion sections created by the lock variable *TSL*.

Notice that a structural definition of mutex bodies will identify no mutex bodies in this function. The only `lock/unlock` pair that might qualify as a mutex body are the statements $L_1$ and $U_3$ (lines 6 and 48 respectively). However, the presence of other `lock` and `unlock` operations in between these statements forces the compiler to disregard this pair as a valid mutex body.

Despite the irregular locking pattern present in this code fragment, it is possible to identify sections that will always execute under the protection of the *TSL* variable. A closer inspection of the code reveals that the only statement that executes without lock protection is the busy wait statement $S_1$ (line 31).

Informally, we modify every `lock` or `unlock` node for lock variable $L$ so that they contain a definition and a use for $L$. All the other nodes in the graph are modified to contain a use for lock variable $L$. To determine whether or not a flow graph node $n$ is protected by lock $L$ we compute reaching definition information for the use of $L$ at $n$. If at least one of the reaching definitions comes from an `unlock` node or if there are no reaching definitions, then node $n$ is not protected by lock $L$.

```
#define NPROCS 5
#define DONE −1

int PopWork(TaskElement *task)
{
    L1 ⇒ lock(TSL);

    while (TaskStackTop == 0) {
        if (++NumWaiting == NPROCS) {
            /* All the threads are waiting for work.
             * We are done.
             */
            lock(pause_lock);
            pause_flag = 1;
            unlock(pause_lock);

            U1 ⇒ unlock(TSL);
            return DONE;
        } else {
            if (NumWaiting == 1) {
                lock(pause_lock);
                pause_flag = 0;
                unlock(pause_lock);
            }

            U2 ⇒ unlock(TSL);

            /* Wait for work. This is the only
             * statement not protected by TSL.
             */
            S1 ⇒ while (!pause_flag) ; /* busy-wait */

            L2 ⇒ lock(TSL);

            if (NumWaiting == NPROCS) {
                U3 ⇒ unlock(TSL);
                return DONE;
            }
            −−NumWaiting;
        }
    } /* while task-stack empty */

    /* Pop a piece of work from the stack */
    TaskStackTop−−;
    task−>left  = TaskStack[TaskStackTop].left;
    task−>right = TaskStack[TaskStackTop].right;

    U3 ⇒ unlock(TSL);

    return 0;
}
```

Figure 3.5: Locking pattern in function *PopWork()*.

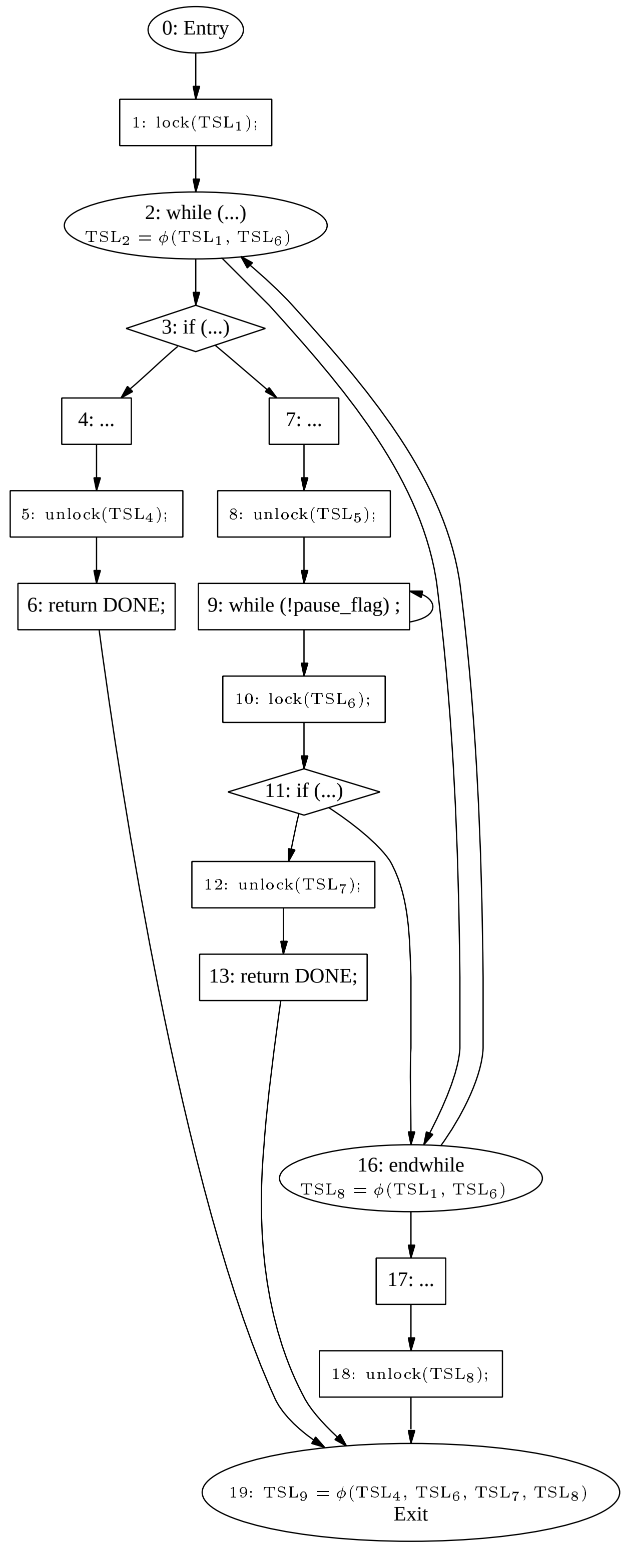



Figure 3.6: Partial SSA form for function *PopWork()*.

The process is illustrated in Figure 3.6. For simplicity, the graph only shows the SSA information related to the lock variable *TSL*. Consider, for instance, node 7. A use of *TSL* in that node can be reached by definitions $TSL_1$ and $TSL_6$. Since both definitions come from a `lock` operation, we conclude that node 7 is protected by the lock *TSL*. Similarly, if we compute reaching definition information for node 9, we conclude that the only definition for *TSL* that can reach it is $TSL_5$. Since $TSL_5$ comes from an `unlock` operation, node 9 is not protected by the lock.

### Detecting Mutex Structures

The detection of mutex structures is reduced to the problem of computing reaching definitions for the lock variables in the program. The Concurrent Control Flow Graph (CCFG) for the program is modified so that:

1. every graph node contains a use for *each* lock variable in the program,
2. every `lock` and `unlock` node for lock variable $L$ contains a definition for $L$, and
3. for each lock variable $L$ the entry node of the graph is assumed to contain an `unlock(L)` operation (this assumption can be overridden using call graph information).

**Definition 3.8 (Lock-protected nodes)** We say that a flowgraph node $b$ is lock-protected by lock $L$ if, and only if, the use of $L$ at $b$ is only reached by definitions of $L$ in `lock(L)` nodes. Therefore, if at least one of those sequential reaching definitions comes from an `unlock(L)` node, then $b$ is not protected by $L$. □

Mutex bodies are defined in terms of lock-protected nodes. For instance, in Figure 3.7(a), the call to $a()$ at line 4 is protected by lock $L$ because it is only reached by the `lock` operation at line 1 and the `lock` operation at line 7. In general, a mutex body is a multiple-entry, multiple-exit region of the graph that encompasses all the flowgraph nodes that are reached by a common set of `lock` nodes. In contrast, previous work (Krishnamurthy and Yelick 1996;

```
lock(L);
while (expr) {

    a();
    unlock(L);
    b();
    lock(L);
    c();
}

unlock(L);
```

(a) Original program. $a()$ and $c()$ are protected by $L$. $b()$ is not.

```
lock(L1);
while (expr) {
    L5 = φ(L1, L3);
    a();
    unlock(L2);
    b();
    lock(L3);
    c();
}
L6 = φ(L1, L3);
unlock(L4);
```

(b) SSA form for the program. $b()$ is not protected because it is reached by an `unlock` operation.

Figure 3.7: Detecting irregular mutex structures in a parallel program.

Masticola and Ryder 1993) has treated mutex bodies as single-entry, single-exit regions.

**Definition 3.9 (Mutex body)** Given a lock variable $L$ and a set of `lock(L)` nodes $N = \{n_1, n_2, \ldots, n_r\}$ known as the *lock nodes*, a *mutex body* $B_L(N) = \{b_1, b_2, \ldots, b_s\}$ is a set of nodes such that:

1. Every node in $\{b_1, b_2, \ldots, b_s\}$ is reached by at least one node $n_i \in N$.

2. There exists at least one node $b_i \in B_L(N)$ that is reached by *all* the nodes in $N$.

3. For every node $n_i \in N$, there exists at least one node $x_i =$ `unlock`$(L)$ such that $x_i$ is reached by $n_i$. All the `unlock`$(L)$ nodes are known as the *unlock nodes* of the mutex body.

4. No node $n_i \in B_L(N)$ can be a `lock(L)` node. □

The first two conditions establish that the nodes in a mutex body must be related in two ways. First, all the nodes in the body must be reached by a common set of `lock(L)` nodes. Second, all the lock nodes must reach at least one common node in the mutex body. Without this restriction, the analysis would consider two disjoint sets of nodes to be the same mutex body.

This clearly makes no sense because they have nothing in common. The third condition defines the exit points of a mutex body. There must be a "way out" of the mutex body from every entry point.

Finally, the fourth condition explicitly excludes lock nodes from the mutex body. This is an important distinction because of the serialization semantics imposed by `lock` operations. A fundamental property of mutex bodies is that given two nodes $a$ and $b$ in two different mutex bodies for the same lock variable, $a$ and $b$ *cannot* execute concurrently. If the lock nodes were considered part of the mutex body, the compiler would think that two concurrent threads can never execute different `lock(L)` nodes at the same time. This is incorrect and therefore not allowed.

Subsequent to this work, Hendren (Hendren 2000) proposed an alternative definition of mutex bodies. For every `lock(L)` node $n$, all the nodes reachable from $n$ are marked in one color. For every `unlock(L)` node $x$, all the nodes reachable from $x$ are marked in another color. The mutex body is the set of nodes that are marked in both colors. This is a much simpler alternative that should lead to more efficient implementations of mutex synchronization analysis.

**Definition 3.10 (Mutex structure)** A mutex structure $M_L$ for lock variable $L$ is the set of all the mutex bodies $B_L(N)$ in the program. □

Mutex structures are detected using sequential reaching definition information for each lock variable $L$. Nodes that are only reached by definitions of $L$ coming from `lock(L)` nodes are protected by $L$. Nodes that can be reached by at least one `unlock(L)` node are not protected by $L$. Using this information Algorithm 3.5 builds an initial set of mutex for each individual `lock(L)` node in the graph. It then refines this initial set by merging mutex bodies with common nodes (see Algorithm 3.5).

We illustrate the process using the SSA form for the sample program in Figure 3.7(b). For simplicity, assume that each line of the program corresponds to a node in the program's flowgraph. The mutex structure for lock $L$ initially contains one mutex body for each `lock(L)` node. In this case there are two mutex bodies for $L$: $B_L(\{1\})$ and $B_L(\{7\})$. Node 1 defines $L_1$ while node 7 defines $L_3$ (Figure 3.7(b)).

Using reached-uses information for definitions $L_1$ and $L_3$ we determine which nodes are reached by each `lock` operation. Consider for instance the node holding the call to $a()$ (node 4). The use of $L$ at node 4 can be reached by definitions $L_1$ and $L_3$. Since both definitions come from `lock(L)` nodes, node 4 is added to both mutex bodies for $L$. Now consider the call to $b()$ at node 6. The use of $L$ at this node can be reached by definition $L_2$ which is an `unlock(L)` node. Therefore, node 6 is not protected and it is not added to any mutex body.

Proceeding in this fashion for all the nodes in the reached-uses set for $L$, Algorithm 3.5 produces two mutex bodies for $L$ (underlined node numbers represent unlock nodes in the mutex body): $B_L(\{1\}) = \{2, 3, 4, \underline{5}, 9, 10, \underline{11}\}$ and $B_L(\{7\}) = \{8, 9, 10, \underline{11}, 2, 3, 4, \underline{5}\}$.

Notice that these two mutex bodies have several nodes in common. Therefore, it is possible to merge them into one mutex body. The resulting mutex structure for $L$ for the program in Figure 3.7(a) contains only one mutex body: $B_L(\{1, 7\}) = \{2, 3, 4, \underline{5}, 8, 9, 10, \underline{11}\}$.

### 3.3.2 Validating Mutex Synchronization

The framework described in the previous section can be used as a validation tool in a compiler. Using this analysis, a compiler can detect irregularities like lock tripping, deadlock patterns, incomplete mutex bodies, dangling `lock` and `unlock` operations and partially protected code (i.e., code that may not always execute under the protection of a lock).

In this section we describe several different illegal locking patterns that can be incorporated into the compiler as compile-time warnings. We say that a `lock(L)` node $n$ reaches another node $m$ if and only if the set of reaching definitions for the use of $L$ at $m$ includes the definition in node $n$.

#### Lock Tripping

We say that a lock has been *tripped over* if the same thread tries to acquire it more than once without releasing it first. This is important to detect because in some systems lock tripping can cause the program to deadlock.

**Algorithm 3.5** Identification of mutex structures.

INPUT: A CCFG $G = \langle N, E, Entry_G, Exit_G \rangle$ in CSSA form, a set $L = \{L_1, L_2, \ldots, L_m\}$ containing all the lock variables used in the program
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 $L_i$.

```
Compute sequential reaching definitions for G.
/* Find candidate mutex bodies and mutex structures. */
foreach lock variable L_i do
  M_i ← ∅
  foreach flowgraph node n such that n = lock(L_i) do
    create mutex body B_{L_i}({n}) = ∅ and add it to M_i
  end for
end for
/* Determine nodes protected by each lock. In this phase mutex bodies are single-node sets. */
foreach mutex structure M_i do
  foreach mutex body B_{L_i}({n}) ∈ M_i do
    d ← definition of L_i in n
    if no node in SeqReachedUses(d) is an unlock(L_i) node then
      disregard B_{L_i}({n})
    else
      foreach use u ∈ SeqReachedUses(d) do
        node ← node(u)
        protected ← TRUE
        foreach definition d ∈ SeqReachingDefs(u) do
          if node(d) is unlock(L_i) then
            protected ← FALSE
          end if
        end for
        if protected then
          add node to mutex body B_{L_i}({n})
        end if
      end for
    end if
  end for
end for
/* Merge mutex bodies that have common nodes. Lock nodes can now have more than one node. */
foreach mutex structure M_i do
  foreach mutex body B^1_{L_i}(N_1) ∈ M_i do
    foreach mutex body B^2_{L_i}(N_2) ∈ M_i do
      if B^1_{L_i}(N_1) ⋂ B^2_{L_i}(N_2) ≠ ∅ then
        B_{L_i}(N_1 ⋃ N_2) ← B^1_{L_i}(N_1) ⋃ B^2_{L_i}(N_2)
        remove B^1_{L_i}(N_1) and B^2_{L_i}(N_2) from M_i
      end if
    end for
  end for
end for
return {M_1, M_2, ..., M_m}
```

```
if (expr) {
    lock(L1);
    ...
} else {
    ...
    lock(L2);
}
L3 = φ(L1, L2);
...
lock(L4);
...
unlock(L);
```

(a) Lock $L$ *will* be tripped at runtime.

```
lock(L1);
...
if (expr) {
    unlock(L2);
    ...
}
L3 = φ(L1, L2);
...
lock(L4);
```

(b) Lock $L$ *may* be tripped at runtime.

Figure 3.8: Some lock tripping scenarios.

Let $L$ be a lock variable and $n$ be a `lock(L)` node. Recall that $n$ contains both a definition and a use for $L$. Suppose that $n$ is reached by other `lock(L)` nodes (Figure 3.8)[1]. If all the definitions come from other `lock(L)` nodes (Figure 3.8(a)), the program is guaranteed to trip over lock $L$ at runtime. If only some definitions come from other `lock(L)` nodes, the program may or may not trip over lock $L$ (Figure 3.8(b)). Depending on the runtime semantics of lock tripping, a compiler may warn the user about the potential problem.

### Deadlock

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

Two different deadlock scenarios are illustrated in Figure 3.9. Both programs launch two threads that satisfy the deadlock requirement described previously. The program in Figure 3.9(a) may or may not deadlock because the mutex body for $M$ in $T_1$ is not always executed. However, the program in Figure 3.9(b) is likely to deadlock because both threads will execute the

[1] The subscripts in the figure refer to SSA numbering. They do not represent different variables.

mutex bodies for $L$ and $M$ for every execution of the program.

Notice that even if these conditions hold, the program may or may not deadlock at runtime. Other conditions like the scheduling of threads or additional synchronization might prevent deadlock situations. A comprehensive deadlock analysis is beyond the scope of our research. Masticola developed techniques that deal specifically with static deadlock detection (Masticola and Ryder 1993).

```
cobegin
    T1: begin
        ...
        lock(L);
        ...
        if (expr) {
            lock(M);
            ...
            unlock(M);
        }
        ...
        unlock(L);
    end

    T2: begin
        ...
        lock(M);
        ...
        lock(L);
        ...
        unlock(L);
        ...
        unlock(M);
    end
coend
```

```
cobegin
    T1: begin
        ...
        lock(L);
        ...
        lock(M);
        ...
        unlock(M);
        ...
        unlock(L);
    end

    T2: begin
        ...
        lock(M);
        ...
        lock(L);
        ...
        unlock(L);
        ...
        unlock(M);
    end
coend
```

Figure 3.9: Some deadlock scenarios.

### Other Locking Irregularities

**Incomplete mutex bodies**. Let $B_L(n)$ be a partially built mutex body for $L$ such that no node in $B_L(n)$ is an `unlock(L)` node. At runtime, if lock $L$ is acquired at $n$, it will not be released. In the presence of incomplete mutex bodies, the compiler may still choose to regard incomplete mutex bodies as complete when optimizing. Nodes that belong to incomplete mutex bodies are still protected by the lock. Optimizations that target mutual exclusion synchronization might be applied provided that they do not require the existence of exit nodes in the mutex body.

**Dangling unlock operations**. Let $x$ be an unlock node for $L$ such that the set of reaching definitions for $L$ at $x$ does not include a `lock(L)` node. This indicates that the calling thread is releasing a lock that it has not acquired. Although releasing an unheld lock might not have consequences at runtime, it indicates a problem with the synchronization structure of the program.

**Partially protected nodes**. Let $b$ be a flowgraph node and $L$ be a lock variable. The framework for building mutex structures guarantees that the set of reaching definitions $RD$ for the use of $L$ at $b$ is not empty.

If all the definitions in $RD$ come from `unlock(L)` nodes, then $b$ is never protected. Conversely, if all the definitions in $RD$ come from `lock(L)` nodes, node $b$ is always protected. However, if some definitions in $RD$ come from a mix of `lock(L)` and `unlock(L)` nodes, then $b$ is only partially protected because it will only be protected on certain executions of the program.

A mutex body with partially protected nodes is said to be an *impure* mutex body. A mutex structure containing *impure* mutex bodies is also considered an *impure* mutex structure and may indicate a possible synchronization problem in the input program.

**Unprotected shared variable references**. Using concurrent reaching-definition information (Algorithm 5.1) it is possible to determine whether all the reaching definitions for a given shared variable use come from mutex bodies in the same mutex structure.

For instance, in the code fragment in Figure 3.10(d) variable $a$ is read and modified by the three threads in the program. Threads $T_1$ and $T_2$ protect the access to $a$ using lock $L$. However, thread $T_0$ does not. Using the concurrent reaching-definition algorithm developed in Section 5.2 the compiler can determine that at least one of the reaching definitions for

$a$ in thread $T_0$ comes from within a mutex body. Since the reference to $a$ made by $T_0$ is not protected and the other concurrent references are, then the compiler can issue a message warning the programmer about the mismatch.

The code fragments shown in Figure 3.10 illustrate each of the locking irregularities previously described.

### 3.3.3 Event Synchronization

Event synchronization imposes execution precedence between related `set` and `wait` nodes. Precedence between `set` and `wait` nodes will also establish precedence for other nodes in the program. Intuitively, nodes preceding the `set` node will execute before nodes after the `wait` node.

The method developed by Lee *et al.* (Lee et al. 1997b) provides a conservative approximate solution to the problem of finding the guaranteed ordering between nodes in the CCFG. In general this problem has been shown to be co-NP hard (Netzer and Miller 1990). For reference, we include their algorithm as Algorithm 3.6.

For each node $n$ in the CCFG of the program, Algorithm 3.6 computes $prec(n)$, the set of nodes guaranteed to execute before $n$. Notice that this particular algorithm has some limitations on the types of programs that it can analyze (Lee et al. 1997b):

1. The body of a sequential loop may not contain the `cobegin/coend` construct.

2. Parallel loops may not contain `set/wait` constructs.

### 3.3.4 Barrier Synchronization

Similar to event-based synchronization, barriers impose ordering constraints in a parallel program. To gather non-concurrency information from barrier synchronization in the program we use the analysis developed by Jeremiassen and Eggers (Jeremiassen and Eggers 1994). This analysis was developed

```
cobegin
    T0: begin
        ...
        lock(L1);
        ...
        /* These statements are
         * protected by L but the lock
         * is never released. */
        ...
    end

    T1: ...
coend
```

(a) Incomplete mutex bodies.

```
cobegin
    T0: begin
        ...
        /* There is no corresponding
         * lock(L) operation.
         */
        unlock(L1);
        ...
    end

    T1: ...
coend
```

(b) Dangling unlock operations.

```
cobegin
    T0: begin
        if (expr) {
            lock(L1);
        }
        ...
        /* These statements may or
         * may not be protected
         * depending on 'expr'
         */
        ...
        if (expr) {
            unlock(L2);
        }
    end

    T1: ...
coend
```

(c) Partially protected nodes (impure mutex bodies).

```
a = 0;
cobegin
    T0: begin
        /* These references to a
         * are not protected by lock L
         */
        a = a + 5;
    end

    T1: begin
        lock(L);
        a = b + 3;
        unlock(L);
    end

    T3: begin
        lock(L);
        print(a);
        unlock(L);
    end
coend
```

(d) Unprotected shared variable references.

Figure 3.10: Locking irregularities.

**Algorithm 3.6** Guaranteed partial execution ordering.

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

/* Fold loop bodies into a representative node. */
/* $Loop(n)$ is a function that returns the set of nodes in a loop whose header is $n$. */
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 $\wedge m \neq n\}$
  $E' \leftarrow (E_f \cup E_s) - \{(m, n) : m, n \in N \wedge (m \notin N' \vee n \notin N')\}$
**foreach** $n \in N'$ **do**
  $prec(n) \leftarrow \emptyset$
**end for**
Initialize work queue $Q$ with the immediate successors of $Entry_G$
**while** $Q \neq \emptyset$ **do**
  Remove some node $n$ from $Q$
  $prec_{old} \leftarrow prec(n)$
  **if** $n$ is `coend` **then**
    $prec_f(n) \leftarrow \bigcup_{(m,n) \in E_{ct}} prec(m) \cup \{n\}$
  **else**
    $prec_f(n) \leftarrow \bigcap_{(m,n) \in E_{ct}} prec(m) \cup \{n\}$
  **end if**
  $prec_s \leftarrow \bigcap_{(m,n) \in E_s} prec(m) \cup \{n\}$
  $prec(n) \leftarrow prec_f(n) \cup prec_s(n)$
  **if** $prec_{old} \neq prec(n)$ **then**
    Put immediate control flow and synchronization successors of $n$ in $Q$
  **end if**
**end while**
**foreach** $n \in N - N'$ **do**
  /* $header(n)$ is a function that returns the header node */
  /* of the outermost loop enclosing $n$ */
  $prec(n) \leftarrow prec(header(n))$
**end for**

for explicitly parallel programs that conform to the SPMD (Single-Program Multiple-Data) model which is compatible to the `parloop` model used in this thesis. In their analysis barriers are assumed to be global: when a thread reaches a barrier it must wait until *all* the other threads in the program cross the same barrier.

The barrier analysis algorithm divides the program into a set of non-concurrent phases. This information is used later on to disregard memory conflicts between nodes in different phases. In what follows we have adapted some of the notation developed in (Jeremiassen and Eggers 1994) to use flowgraph nodes instead of statements.

We denote barrier nodes $B(i, x)$, where $i$ is a unique integer identifying the barrier call site and $x$ is the name of the barrier variable being crossed (Figure 3.11, adapted from Jeremiassen's paper (Jeremiassen and Eggers

```
cobegin {
  T0: begin   /* Workers */
    parloop (i, 0, N − 1) {
      while (!converged) {
        barrier(a, N);
        partA();

        barrier(b, N);
        partB();

        barrier(c, N);
      }
    }
  end

  T1: begin   /* Master */
    while (converged == 0) {
        produceA();
        barrier(a, N);

        produceB();
        barrier(b, N);

        converged = has_converged();
        barrier(c, N);
    }
  end
}
```

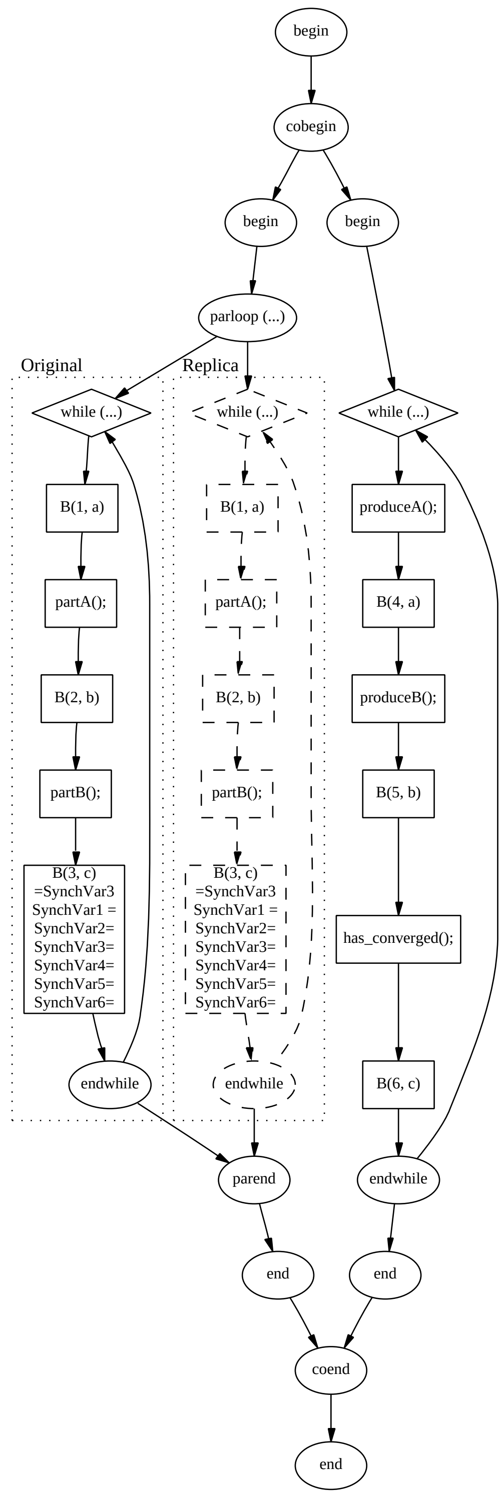



Figure 3.11: An example of barrier synchronization.

1994)). Barrier nodes define *process segments.* A process segment is the set of all the flowgraph nodes along barrier free control paths between one barrier node $B(i, x)$ and another barrier node $B(j, y)$. Process segments are denoted using the barrier call sites at either end of the segment: $(B_i, B_j)$. There is an implicit barrier at the start of the program denoted $S$.

A *phase* of the program is the set of process segments that may execute concurrently between two global barriers. The goal of the barrier analysis algorithm is to divide the flowgraph into a set of process segments and partition these segments into a set of phases. Nodes in segments from two different phases cannot execute concurrently.

There are two stages to the algorithm. The first stage divides the program into sets of process segments by computing which other barriers can be reached from each barrier. This is similar to the problem of matching `lock` and `unlock` operations described in Section 3.3.1 but they use a different approach. For each barrier node $B(n, x)$ in the CCFG a variable $SynchVar_n$ is created. Then, each barrier node $B(n, x)$ is modified so that right after the barrier call the node contains a use of variable $SynchVar_n$ followed by a definition of *all* the variables $SynchVar_i$.

The next step is to determine which of the $SynchVar_i$ variables are *live* at the end of each barrier node. If variable $SynchVar_j$ is live at barrier node $B(i, x)$ (i.e., its value is going to be used again along some program path starting at that node), then we create the process segment $(B_i, B_j)$.

We illustrate this process using the program in Figure 3.11. Consider the barrier node $B(3, c)$. We modify the node so that it contains a use of variable $SynchVar_3$ followed by definitions of six other $SynchVar$ variables used for this program. Variable $SynchVar_1$ is live at node $B(3, c)$ because its value is used again at node $B(1, a)$. Therefore, $(B_3, B_1)$ is a process segment of the program. Proceeding in this fashion we obtain the complete set of process segments for the program: $(S, B_1)$, $(S, B_4)$, $(B_1, B_2)$, $(B_2, B_3)$, $(B_3, B_1)$, $(B_4, B_5)$, $(B_5, B_6)$ and $(B_6, B_4)$.

The second stage of the algorithm partitions the process segments into non-concurrent phases using a work queue approach. The initial set of phases is created by assuming that all the process segments that start at the same

| | **Initial state** | **Iteration 1** | **Iteration 2** | **Final state** |
|---|---|---|---|---|
| Phase 1 | $\{(S,B_1),(S,B_4)\}$ | $\{(S,B_1),(S,B_4)\}$ | $\{(S,B_1),(S,B_4)\}$ | $\{(S,B_1),(S,B_4)\}$ |
| Phase 2 | $\{(B_1,B_2)\}$ | $\{(B_1,B_2),(B_4,B_5)\}$ | $\{(B_1,B_2),(B_4,B_5)\}$ | $\{(B_1,B_2),(B_4,B_5)\}$ |
| Phase 3 | $\{(B_2,B_3)\}$ | $\{(B_2,B_3)\}$ | $\{(B_2,B_3),(B_5,B_6)\}$ | $\{(B_2,B_3),(B_5,B_6)\}$ |
| Phase 4 | $\{(B_3,B_1)\}$ | $\{(B_3,B_1)\}$ | $\{(B_3,B_1)\}$ | $\{(B_3,B_1),(B_6,B_4)\}$ |
| Phase 5 | $\{(B_4,B_5)\}$ | | | |
| Phase 6 | $\{(B_5,B_6)\}$ | $\{(B_5,B_6)\}$ | | |
| Phase 7 | $\{(B_6,B_4)\}$ | $\{(B_6,B_4)\}$ | $\{(B_6,B_4)\}$ | |

Figure 3.12: Partition of process segments into phases for the program in Figure 3.11.

barrier call site and end at barrier nodes that cross the same variable can execute concurrently. The initial set of phases is refined in an iterative process by merging phases that can execute concurrently. Each phase $P_i$ is examined so that for each pair of process segments $(B(j,x),B(k,y))$ and $(B(r,z),B(s,y))$ in $P_i$ it creates a new phase with all the phases that start with $B(k,y)$ or $B(s,y)$ in any of their process segments and whose process segments end in the same barrier node. Figure 3.12 illustrates this iterative process applied to the example program in Figure 3.11.

The algorithm stops when the work queue is empty (i.e., no more phases can be merged into a new one). The output of the algorithm is a set of non-concurrent phases $P_1, P_2, \ldots P_m$. Each phase $P_i$ contains a set of process segments which, in turn, delimit sets of CCFG nodes. The data-flow analysis techniques developed in Chapter 4 will use this information to determine whether two arbitrary CCFG nodes can execute concurrently. If nodes $a$ and $b$ belong to process segments from two different phases then they cannot execute concurrently.

## 3.4 Summary

The Concurrent Control Flow Graph (CCFG) is the basic data structure used to analyze and optimize an explicitly parallel program. It describes the control structure of the program as well as memory conflicts and event-based synchronization. We then use the CCFG to gather non-concurrency information. First, the parallel structure of the CCFG determines an initial set of graph nodes that may execute concurrently (Algorithm 3.2).

The initial set of concurrent flowgraph nodes is then refined by analyzing the synchronization structure of the program (Section 3.3). We have developed a new technique to analyze non-concurrency for mutex synchronization that can handle locking patterns not supported by existing techniques. This is a significant improvement that allows the analysis of more complex mutual exclusion synchronization patterns in explicitly parallel programs. We also adapt existing techniques that analyze `set/wait` and `barrier` synchronization.

Non-concurrency techniques are important in the context of an optimizing compiler for explicitly parallel programs. Since the problem of analyzing non-concurrency is orthogonal to the data-flow framework, as new techniques are discovered they can be readily incorporated into the compiler with little or no modifications to the overlying data-flow framework. In the next chapter we develop an SSA-based data-flow framework that uses the synchronization analyses developed in this chapter to determine whether some memory conflicts can be disregarded because of synchronization constraints.

# Chapter 4

# The CSSAME Form

This chapter describes the CSSAME form, a data-flow framework for analyzing explicitly parallel programs. The CSSAME form builds on and extends the CSSA form (Lee et al. 1997b) which is described in Section 4.1. Section 4.2 introduces the extensions necessary to build the CSSAME form. The extensions allow the framework to handle parallel loops[1], mutual exclusion and barrier synchronization in explicitly parallel programs.

Algorithms and time complexity analyses are included in the discussion. We point out that algorithmic design decisions have been made to favor clarity of presentation, they should not be an indication of how an actual implementation should be organized. In particular, an implementation might decide to perform all the $\pi$ rewriting actions of Sections 4.2.4 and 4.2.5 prior to the placement of conflict edges to simplify the task of placing $\pi$ functions in the first place.

## 4.1 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 $\phi$ function is introduced to resolve the ambiguity. The $\phi$ function merges all the incoming reaching

[1]In recent work, Lee *et al.* have independently incorporated parallel loops into their framework (Lee et al. 1999).

definitions to create a new definition for the variable (Cytron et al. 1991). In a parallel program, the single assignment property is disrupted by the presence of concurrent definitions to the variable because definitions made in concurrent threads may be observed at the thread reading the shared variable. The CSSA framework solves this ambiguity with $\pi$ functions. A $\pi$ function merges the definitions coming from the current thread via control paths and other concurrent threads via conflict edges.

This section describes the algorithms needed to build the CSSA form as described in (Lee et al. 1997b). Algorithm 4.1 computes the CSSA form of a program. The algorithms to place $\phi$ functions and build factored use-def chains compute the sequential SSA form (Wolfe 1996). Note that all the algorithms in this section are unmodified versions of the original references. They are only included to facilitate an implementation of the CSSAME framework and simplify the discussion of the complexity analysis of the CSSAME algorithm.

**Algorithm 4.1** Build the CSSA form.

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

1: Find guaranteed execution ordering using Algorithm 3.6.
2: Build sequential SSA form using Algorithms 4.2 and 4.3.
3: Place $\pi$ functions using Algorithm 4.4.

### 4.1.1 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 (Wolfe 1996). Algorithm 4.2 adds $\phi$ functions to the graph and Algorithm 4.3 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:

$child(n)$ is the set of dominator children for node $n$.

$succ(n)$ is the set of immediate successors of node $n$.

$whichPred(n \rightarrow m)$ is an index telling which immediate predecessor of $m$ corresponds to the control edge from $n$.

$DF(n)$ is the dominance frontier for node $n \in G$.

$D(v)$ is the set of nodes in $G$ that contain a definition for variable $v$.

$Symbols$ is the set of variables used in the program.

**Algorithm 4.2** Place $\phi$ functions.

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

```
foreach n ∈ N do
  inWork(n) ← ⊥
  added(n) ← ⊥
end for
workList ← ∅
foreach v ∈ Symbols do
  foreach n ∈ D(v) do
    workList ← workList ∪ {n}
    inWork(n) ← v
  end for
  while workList ≠ ∅ do
    Remove some node n from workList
    foreach w ∈ DF(n) do
      if added(w) ≠ v then
        Add φ function for v at w
        added(w) ← v
        if inWork(w) ≠ v then
          workList ← workList ∪ {w}
          inWork(w) = v
        end if
      end if
    end for
  end while
end for
```

### 4.1.2 Placing $\pi$ Functions

The final phase of the CSSA algorithm traverses the graph placing $\pi$ functions at every node that contains one or more conflicting variable uses. Algorithm 4.4 adds the required $\pi$ functions to the graph. The basic principle is straightforward, if a shared variable is used in a node and there exist concurrent definitions for that variable, a $\pi$ function is needed in the node where the variable is read.

Recall from section 3.1 that nodes with conflicting use references for variable `v` have one `DU(v)` conflict edge for each definition of `v` in concurrent threads. Furthermore, there will be a definition of `v` coming from the incoming

**Algorithm 4.3** Build FUD chains.

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

```
foreach v ∈ Symbols do
  currDef(v) ← ⊥
end for
call search(Entry_G)

procedure search(x)
foreach variable use or def or φ function r ∈ x do
  m ← variable referenced at r
  if r is a use then
    chain(r) ← currDef(m)
  else if r is a def or a φ function then
    saveChain(r) ← currDef(m)
    currdef(m) ← r
  end if
end for

foreach y ∈ succ(x) do
  j ← whichPred(x → y)
  foreach φ function r in y do
    m ← variable referenced at r
    φ − chain(r)[j] ← currDef(m)
  end for
end for

foreach y ∈ child(x) do
  call search(y)
end for

foreach variable use or def or φ function r ∈ x in reverse order do
  m ← variable referenced at r
  if r is a def or a φ function then
    currDef(m) ← saveChain(r)
  end if
end for
```

control edge. Therefore, Each $\pi$ function has $n + 1$ arguments; the unique incoming control flow edge and the $n$ incoming conflict edges. As we will discuss later in this document, some of these arguments to a $\pi$ function may be proven redundant because of synchronization operations in the program.

### 4.1.3 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 3.6). 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

**Algorithm 4.4** Place $\pi$ 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

```
foreach b ∈ N do
  foreach DU conflict edge e = (a, b) do
    v ← variable defined in a
    if b does not have a π function for v then
      Insert a new π function for v in b
      u ← conflicting use of v in b
      π(v)[0] ← chain(u)
    end if
    if n ∉ prec(s) then
      d ← conflicting def of v in s
      append d to π(v)
    end if
  end for
end for
```

(Algorithms 4.2 and 4.3). This phase computes the SSA form in $O(r^3)$ time, where $r$ is the maximum of the number of nodes ($|N|$), number of control edges ($|E_f|$), number of assignments and number of variable references in the program (Brandis and Moessenboeck 1994; Cytron et al. 1991). Note that it is possible to place $\phi$ function using the linear time algorithms in (Johnson et al. 1994) and (Sreedhar and Gao 1995). We use the algorithms from (Wolfe 1996) solely because they are easier to implement.

The third phase of the computation of the CSSA form places $\pi$ functions at the concurrent join nodes of the graph (Lee et al. 1997b). By examining the $\pi$ placing algorithm (Algorithm 4.4) 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 $\phi$ functions. If the traditional $\phi$ placing algorithms are used, then the CSSA form can be computed in $O(r^3)$ time.

## 4.2 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 (Lee et al. 1997b). 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 to the lock variable and `unlock` operations only write to it, an implementation should create $\pi$ functions for every `lock` node in the graph.

The key observation that gives rise to the CSSAME form is that $\pi$ 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 $\pi$ functions inside mutex bodies (Sections 4.2.2 and 4.2.3). This analysis is important because it allows the removal of redundant conflict edges which in turn allows the optimizer to safely apply more aggressive transformations and generate faster code. Both removal conditions can be implemented as predicates called by the compiler when analyzing mutex bodies.

### 4.2.1 Parallel Loops

Parallel loops are treated similarly to `cobegin/coend` structures. The loop body is replicated to allow the parallel loop to be considered like a `cobegin/coend` structure with two identical bodies. This is enough for the purposes of this analysis because we are only interested in determining whether there is a memory referencing conflict or not. It is not necessary to determine how many threads participate in the conflict. Knowing that there is at least two threads in conflict is enough.[2] A similar approach is taken in (Krishnamurthy and Yelick 1996) and (Lee et al. 1999). The process of adding $\pi$ functions does not need to be modified to handle parallel loops because every node in the loop body is concurrent with its replica and with every other node inside the parallel loop.

All the transformations to $\pi$ functions due to synchronization are performed

[2] This of course may have to be revised if other analyses need more specific information about the conflict.

```
parloop (i, 1, N) {
    a = ...;
    ... = a + 4;
}
```

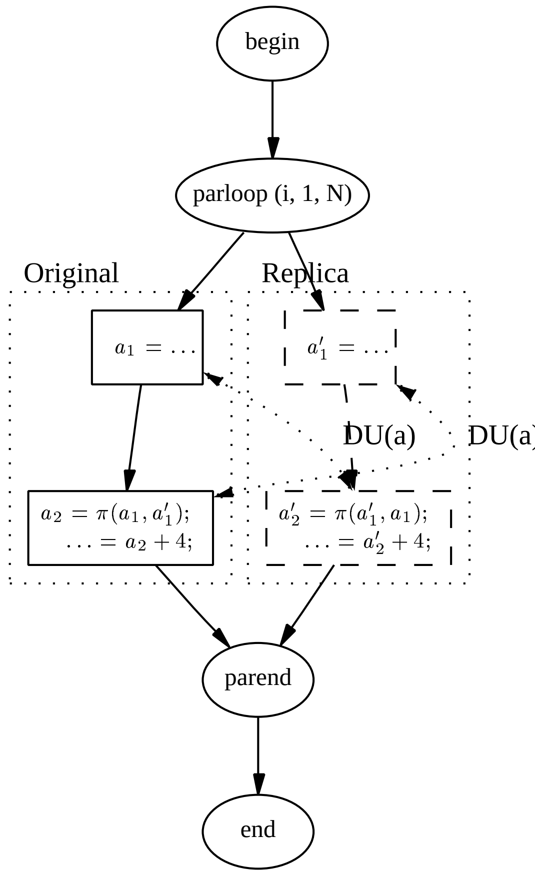



Figure 4.1: $\pi$ functions inside a parallel loop.

on the original loop body. For instance, consider the code fragment in Figure 4.1. The conflict analysis algorithm has determined that there is a conflict between the node that defines $a$ and the node that uses $a$ to compute $a + 4$. Notice that the $\pi$ function generated for the second node contains the arguments $a_1$ and $a'_1$. The first $a_1$ is the definition inherited via the control path. The second $a'_1$ is the definition coming from the loop body's replica. This replica represents one of the $N$ concurrent threads executing the body of the parallel loop.

```
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 4.2: Removing memory conflicts.

### 4.2.2 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 4.2(a) where definition $a_1$ in thread $T_0$ is overridden by definition $a_2$ in the same thread. Therefore, the read reference $a_3$ in thread $T_1$ can only be reached by definition $a_2$.

**Definition 4.1 (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 (Aho et al. 1986). □

**Theorem 4.1 (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 $B_L(N) \in M_L$. If $D_a^B$ does not reach any exit node $x \in B_L(N)$ then $D_a^B$ can be removed from all the $\pi$ functions in any other mutex body $B'_L(N') \in M_L$ that have $D_a^B$ as an argument. □

PROOF Let $U_a^{B'}$ be a use of $a$ in $B'_L(N')$. 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. $B_L(N)$ executes to completion before $B'_L(N')$. Even though node $d$ executes 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 one of the exit nodes of $B_L(N)$.

2. $B'_L(N')$ executes to completion before $B_L(N)$. 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 $\pi$ 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. ∎

### 4.2.3 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 4.2(b)). Since $a$ is defined inside the mutex body, definitions made in other mutex bodies are killed by the internal definition of $a$.

**Definition 4.2 (Upward exposure for mutex bodies)** Given a mutex body $B$, a use $U_v^B$ in $B$ for a variable $v$ is *upward-exposed* (Aho et al. 1986) from $B$ if $U_v^B$ may use a definition outside of $B$. □

**Theorem 4.2 (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 $B_L(N) \in M_L$. If $U_a^B$ is not upward-exposed from $B_L(N)$ then the arguments for the $\pi$ function for $a$ coming from any other mutex body $B'_L(N') \in M_L$ can be removed. □

Proof Let $D_a^{B'}$ be a definition for variable $a$ in mutex body $B'_L(N')$. Let $d$

be the node in $B'_L(N')$ that contains the definition $D_a^{B'}$. Let $u$ be the node in mutex body $B_L(N)$ 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. $B_L(N)$ executes to completion before $B'_L(N')$. This means that node $u$ executes before node $d$, therefore $D_a^{B'}$ cannot reach $U_a^B$.

2. $B'_L(N')$ executes before $B_L(N)$. Since $U_a^B$ is not upward-exposed from $B_L(N)$, any definitions of $a$ made before $B_L(N)$ starts executing are guaranteed to be killed by some other definition inside $B_L(N)$. 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 $\pi$ 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. ■

### 4.2.4 Modifying $\pi$ Functions Inside Mutex Bodies

Using the properties of consecutive kills and protected uses inside mutex bodies, we now examine every mutex body of the program trying to remove arguments from each of its $\pi$ functions. Algorithm 4.5 traverses all the mutex bodies in the graph looking for $\pi$ 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 $\pi$ functions inside a mutex body $b$. For each $\pi$ function, each of its arguments $d$ is analyzed for compliance with Theorems 4.1 and 4.2.

   Checking for protected uses is a simple matter of checking whether the control reaching definition for the $\pi$ function is reached by at least one lock node in $N$. This information has already been computed by the mutex structure detection algorithm (Section 3.3.1). Therefore, it can be accessed 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 some exit node (i.e., we check whether there is another definition $d'$ on every path from $d$ to an exit node that kills $d$).

3. Lines 21–25 remove any $\pi$ functions with no arguments for conflicting references.

Examining the nesting structure of the $\pi$ 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 maximum number of nodes that a mutex body can contain, $|\pi|$ is the number of $\pi$ 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 4.1 (Correctness of the $\pi$ rewriting algorithm)** The only arguments from $\pi$ functions removed by Algorithm 4.5 represent memory interleavings that cannot occur at runtime. □

Proof The algorithm only examines $\pi$ functions inside mutex bodies. For each $\pi$ 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 4.1 and 4.2 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 $\pi$ 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 4.5. Hence, this $\pi$ 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). ■

**Algorithm 4.5** Rewrite $\pi$ functions to account for mutual exclusion.

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 π functions to rewrite. */
 2: foreach lock variable L_i do
 3:   foreach mutex body b ∈ MutexStruct(L_i) do
 4:     call rewrite(b)
 5:   end for
 6: end for

 7: /* Examine all the π functions in b. */
 8: procedure rewrite(b)
 9: foreach node n ∈ b do
10:   foreach π function p ∈ n do
11:     v is the variable referenced by p
12:     /* If an argument of the π function p complies with Theorems 4.1 or 4.2, */
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' ∈ MutexStruct(b) then
16:         if (the use of v is not upward-exposed from b) or (d does not reach any 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) ← first argument of p
24:       remove p from n
25:     end if
26:   end for
27: end for
```

### 4.2.5 Modifying $\pi$ Functions Affected by Barriers

Barrier synchronization offer another source of non-concurrency information in parallel programs. Using the barrier analysis algorithm described in Section 3.3.4 it is possible to remove $\pi$-function arguments for some conflict edges that cross phase boundaries. Since nodes in different phases of the program are guaranteed to execute in sequence, some of the conflicts that might exist between these nodes can be eliminated.

Barrier synchronization is "weaker" than mutex synchronization in the sense that it does not serialize the execution of threads. The ordering created by barriers create phases in the execution of the program. Within a phase, threads execute concurrently. Consider for instance the parallel loop in Figure 4.3. If we disregard the presence of the barrier, then both definitions $a_1$ and $a_2$ can reach the use of $a$ ($a_3$) at line 10. However, the presence of the barrier at line 5 guarantees that definition $a_1$ will be killed by all the threads before crossing the barrier. Therefore, $a_1$ cannot reach the use of $a$ at line 10. The same cannot be said about definition $a_2$. Although all threads join at the barrier, we cannot statically determine which thread will be the last to reach the barrier. This means that there are two definitions for variable $a$ that could reach $a_3$: the control reaching definition (i.e., $a_2$, the sequential reaching definition) and the definition made by the last thread to join the barrier ($a_2'$). In general, in the presence of barriers the only arguments that can be removed from a $\pi$ function are those that represent definitions from a different phase and do not reach the $\pi$ function via control edges.

**Theorem 4.3 (Barrier protection)** Let $U_v$ be a conflicting use for shared variable $v$. Let $D_v$ be a definition for $v$ such that $D_v$ reaches $U_v$ via a conflict edge and $D_v$ does not sequentially reach $U_v$. If $D_v$ and $U_v$ are in different phases due to barrier synchronization, then $D_v$ can be removed from the $\pi$ function associated with $U_v$. □

Proof Since $D_v$ reaches via a conflict edge, there is a $\pi$ function associated with $U_v$ that has $D_v$ as one of its arguments. If $D_v$ and $U_v$ are on different phases as determined by barrier synchronization analysis (Section 3.3.4), then they cannot execute concurrently. Furthermore, since $D_v$ does not reach $U_v$ via control edges, it means that there exists at least one other definition for $v$ that kills $D_v$. Since $D_v$ cannot reach $U_v$ via control edges nor conflict edges, it is safe to remove it from the $\pi$ function associated with $U_v$. ■

Algorithm 4.6 rewrites $\pi$ functions to account for barrier synchronization. It assumes that program phases have already been computed (Section 3.3.4). The algorithm traverses all the $\pi$ functions in the program. For every argument $d_i$ of a $\pi$ function $p$ it checks which node contains $d_i$. If the node of $d_i$ is inside a different phase than the node holding $p$ and $d_i$ does not sequentially reach the use associated with $p$, then $d_i$ can be removed from the argument list.

Figure 4.3 shows a program fragment with its CSSAME form partially

```
1   parloop (i, 1, N) {
2       a1 = c1 + 5;
3       ...
4       a2 = a1 + c1;
5       barrier(B, N);
6       ...
7       /* Argument a1' can be safely
8          removed from this π function. */
9       a3 = π(a2, a1', a2');
10      b1 = a3 + 3;
11  }
```

Figure 4.3: Effects of barrier synchronization on $\pi$ functions.

built. The assignment to $b$ in line 10 makes a conflicting use of variable $a$. Hence the $\pi$ function at line 9 contains only two arguments and both come from the same definition ($a_1$ is both the control-reaching and the conflict-reaching definition). The computation of phases for this program will result in two phases, one containing lines $1-4$ and the other one containing lines $6-10$. Therefore, definitions $a_1$ and $a_2$ will be in one phase and use $a_3$ will be in another one. Since definition $a_1$ is killed by $a_2$ and it is in a different phase than the use $a_3$, we can remove the second argument of the $\pi$ function at line 9 because $a_1$ cannot reach this use.

Notice that unlike mutex synchronization, this pruning process will never lead to the elimination of $\pi$ functions. The reason is that inside a parallel loop $\pi$ functions have two arguments coming from the same definition, namely the control reaching definition. The control reaching definition appears twice in the $\pi$ argument list because it reaches the use via control and conflict edges. The argument coming via control edges cannot be eliminated because it is not affected by synchronization and the argument coming via a conflict edge cannot be eliminated because it is not possible to determine which thread was the last one to make that definition. It might be possible to eliminate a $\pi$ function if one could prove that both arguments are always the same value using techniques like value numbering, copy propagation or constant propagation. We have not considered these extensions in this document.

**Algorithm 4.6** Rewrite $\pi$ functions to account for barrier synchronization.

INPUT: A Parallel Flow Graph $G = \langle N, E, Entry_G, Exit_G \rangle$ in CSSA form
OUTPUT: The graph $G$ in CSSA form with $\pi$ functions modified to account for barrier synchronization

/* This algorithm assumes that phases due to barrier */
/* synchronization have already been computed (Section 3.3.4). */

compute sequential reaching definitions (*SeqReachingDefs*)
**foreach** $\pi$-function $p$ **do**
  $u \leftarrow$ use reference associated with $p$
  **foreach** parallel argument $d$ of $p$ **do**
    **if** $node(p)$ and $node(d)$ are in different phases **and** $d \notin SeqReachingDefs(u)$ **then**
      remove $d$ from $p$
    **end if**
  **end for**
**end for**

### 4.2.6 Computing the CSSAME Form

Algorithm 4.7 transforms an explicitly parallel program $P$ to its CSSAME form. The algorithm is a direct extension of the CSSA algorithm (Lee et al. 1997b). 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 3.2. 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 4.1. Once the CSSA form has been computed, $\pi$ functions are modified to account for any mutex and/or barrier synchronization in the program. Notice that it might be possible to compute the CSSAME form directly, without computing the CSSA form first. We decided to use this approach because the analysis needed to remove superfluous synchronization edges is simpler if CSSA is computed first.

**Theorem 4.4 (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 (Lee et al. 1997a; Wolfe 1996). Lemma 4.1 proves that the only

**Algorithm 4.7** Build the CSSAME form.

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

1: Build the CCFG $G$ for $P$ using Algorithm 3.1.
2: Identify mutex structures using Algorithm 3.5.
3: Compute the CSSA form for the graph using Algorithm 4.1.
4: Rewrite $\pi$ functions using Algorithm 4.5.
5: Rewrite $\pi$ functions using Algorithm 4.6.

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. ∎

### 4.2.7 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 $\pi$ functions). As discussed in Chapter 3, the identification of mutex structures can be done in $O(|E_f|)$ 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 ($|E_f|$), number of assignments and number of variable references in the program (Section 4.1.3). Finally, rewriting $\pi$ functions can be done in $O(|N|^3)$ time. Therefore, the CSSAME algorithm has a worst time complexity of $O(|N|^3)$.

## 4.3 Summary

In this chapter we have developed a new data-flow framework for explicitly parallel programs: the CSSAME form. It supports both task and data parallel programs that share memory and synchronize using three types of mechanisms: mutual exclusion, barriers and events.

The CSSAME form represents a significant step towards an integrated analysis framework that can be adapted to support various types of parallel constructs, memory semantics and synchronization constructs. For instance, to add a new type of synchronization mechanism, we only need to gather non-concurrency information due to synchronization and modify the $\pi$

functions appropriately. Different memory semantics can be supported in a similar fashion. Memory conflicts across concurrent threads need only be added if the memory semantics of the target architecture allow such interleaving. For instance, in a release-consistent memory (Keleher et al. 1994) memory conflicts need only be added at synchronization points in the program.

In the following chapter we use the CSSAME framework to optimize parallel programs. We will consider two types of optimization, the adaptation of sequential techniques to the parallel case and the direct optimization of the synchronization structure of a parallel program. Emphasis will be on the optimization of mutual exclusion patterns.

# Chapter 5

# Optimizing explicitly parallel programs

Using the CSSAME form, new optimization opportunities are now possible. This section describes six optimization techniques. The first two are adaptations of well-known sequential optimizations: constant propagation (Section 5.1) and dead code elimination (Section 5.2). The other four are new optimizations specifically designed for explicitly parallel programs: lock picking (Section 5.3), lock-independent code motion (Section 5.4), mutex body localization (Section 5.5) and single-writer multiple-readers code motion (Section 5.5.1). All the mutual exclusion transformations in this chapter assume that the program contains well-formed mutex structures.

## 5.1 Constant Propagation

Lee *et al.* (Lee et al. 1997b) adapted the sequential Sparse Conditional Constant propagation (SCC) algorithm (Wegman and Zadeck 1991) to work with explicitly parallel programs; Concurrent Sparse Conditional Constant propagation (CSCC). We will use the program in Figure 5.1(a) 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 5.1(b) is the original CSSA form without mutual exclusion extensions. Figure 5.2(a) shows the CSSAME form built using the algorithms in Section

4.2. Notice that the CSSAME form has fewer $\pi$ functions than the CSSA form.

```
a = 0;
b = 0;
cobegin
  T0: begin
    lock(L);
    a = 5;
    b = a + 3;
    if (b > 4) {
      a = a + b;
    }
    x = a;
    unlock(L);
  end

  T1: begin
    lock(L);
    a = b + 6;
    y = a;
    unlock(L);
  end
coend
print(x, y);
```

(a) Original program.

```
a1 = 0;
b1 = 0;
cobegin
  T0: begin
    lock(L);
    a2 = 5;
    a3 = π(a2, a6);
    b2 = a3 + 3;
    if (b2 > 4) {
      a4 = π(a2, a6);
      a5 = a4 + b2;
    }
    a7 = φ(a2, a5);
    a8 = π(a7, a6);
    x1 = a8;
    unlock(L);
  end

  T1: begin
    lock(L);
    b3 = π(b1, b2);
    a6 = b3 + 6;
    a9 = π(a6, a2, a5);
    y1 = a9;
    unlock(L);
  end
coend
a10 = φ(a7, a6);
print(x1, y1);
```

(b) CSSA form.

Figure 5.1: Constant propagation example (CSSA).

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 $\pi$ 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 5.2(b) 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 (Wegman and Zadeck 1991). Since we have not modified the CSCC algorithm, the optimizations performed are still correct as proved in (Lee et al. 1997b).

Further optimizations can still be done in this example program. The redundant assignments in Figure 5.2(b) are the result of applying the concurrent constant propagation on the program in Figure 5.2(a). These redundant assignments can be removed using the concurrent dead-code elimination algorithm developed in Section 5.2.

```
a1 = 0;
b1 = 0;
cobegin
  T0: begin
    lock(L);
    a2 = 5;
    b2 = a2 + 3;
    if (b2 > 4) {
      a3 = a2 + b2;
    }
    a4 = φ(a2, a3);
    x1 = a4;
    unlock(L);
  end

  T1: begin
    lock(L);
    b3 = π(b1, b2);
    a5 = b3 + 6;
    y1 = a5;
    unlock(L);
  end
coend
a6 = φ(a4, a5);
print(x1, y1);
```

(a) CSSAME form for program in Figure 5.1(a).

```
a1 = 0;
b1 = 0;
cobegin
  T0: begin
    lock(L);
    a2 = 5;
    b2 = 8;
    a3 = 13;
    a4 = 13;
    x1 = 13;
    unlock(L);
  end

  T1: begin
    lock(L);
    b3 = π(b1, b2);
    a5 = b3 + 6;
    y1 = a5;
    unlock(L);
  end
coend
a6 = φ(a4, a5);
print(x1, y1);
```

(b) Constant propagation using CSSAME.

Figure 5.2: Constant propagation example (CSSAME).

## 5.2 Concurrent Dead Code Elimination

Dead code refers to program statements that have no effect on the program output (Cytron et al. 1991). Although it is not common for the programmer to introduce dead code intentionally, dead code may be generated by optimizing transformations (Aho et al. 1986). We introduce the Concurrent Dead Code Elimination algorithm (CDCE), an extension of the dead code elimination algorithm proposed by Cytron *et al.* (Cytron et al. 1991) 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 (Cytron et al. 1991):

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.

The CDCE algorithm is the same algorithm developed by Cytron *et al.* (Cytron et al. 1991) with the following modifications:

- 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 (Wolfe 1996) by incorporating additional tests for $\pi$ functions when traversing the SSA use-def chains. Concurrent reaching definition information is computed by Algorithm 5.1.

- 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 $\pi$ and $\phi$ 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 4.1) will help the detection of dead code inside mutex bodies.

To show the effects of CDCE, consider the program in Figure 5.1(a) after constant propagation has been performed (Figure 5.2(b)). 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 5.3 shows the result of a dead code pass on the code in Figure 5.2(b).

**Theorem 5.1 (Correctness of the CDCE algorithm)** 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

```
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);
```

Figure 5.3: Concurrent Dead Code Elimination for program in Figure 5.2(b).

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. ■

## 5.3 Lock Picking

Sometimes it is possible to remove synchronization operations from a program without affecting its semantics. For example, mutual exclusion synchronization is unnecessary in a sequential program and can be safely removed. In this section we describe *lock picking*, a transformation that finds and removes superfluous `lock` and `unlock` operations. We say that a mutex body can be *lock-picked* if its lock and unlock nodes can be removed. An

**Algorithm 5.1** 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) ← ⊥
  uses(d) ← ∅
end for
foreach variable use u in the program do
  defs(u) ← ∅
  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) ← 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 φ or a π function, follow the arguments */
if (d is a φ function) or (d is a π function) then
  foreach function argument j do
    call followChain(j, u)
  end for
end if
```

important property of lock picking is that it *does not* need to examine the mutex bodies of the program. Only the lock and unlock nodes are analyzed.

Lock picking uses reaching definition information for all the lock variables to determine whether a mutex body can be lock-picked or not. The algorithm for recognizing mutex bodies developed in Section 3.3.1 modifies the flowgraph so that every `lock(L)` node contains one definition of variable $L$ and a use for each lock variable used in the program (including $L$). As such, the CSSAME form will initially place a $\pi$ function for all the uses of lock variables at each mutex body's lock node. However, if the program contains additional synchronization, it is possible that some of these $\pi$ functions will be removed by the CSSAME $\pi$ pruning phase. Furthermore, in the case of sequential sections of the program, $\pi$ functions will not be placed at all.

The lock picking algorithm (Algorithm 5.2) examines the lock nodes for every mutex body in the program. The decision to lock-pick a mutex body

```
double Sum = 0;
parloop (p, 0, N) {
  ...
  for (i = 0; i < M; i++) {
    S3 = π(S0, S1, S2);
    R3 = π(R0, R1, R2);
    lock(R1);
    for (j = 0; j < M; j++) {
      sum_reduction(A[i][j]);
    }
    unlock(R2);
  }
  ...
}

sum_reduction(double x)
{
  S4 = π(S0, S1, S2)
  R4 = π(R0, R1, R2)
  lock(S1);
  Sum = Sum + x;
  unlock(S2);
}
```

(a) Original CSSAME form.

```
double Sum = 0;
parloop (p, 0, N) {
  ...
  for (i = 0; i < M; i++) {
    S3 = π(S0, S1, S2)
    R3 = π(R0, R1, R2)
    lock(R1);
    for (j = 0; j < M; j++) {
      S4 = π(S0, S1, S2)
      lock(S1);
      Sum = Sum + A[i][j];
      unlock(S2);
    }
    unlock(R2);
  }
  ...
}
```

(b) CSSAME form after inlining and $\pi$ pruning.

```
double Sum = 0;
parloop (p, 0, N) {
  ...
  for (i = 0; i < M; i++) {
    R3 = π(R0, R1, R2)
    lock(R1);
    for (j = 0; j < M; j++) {
      Sum = Sum + A[i][j];
    }
    unlock(R2);
  }
  ...
}
```

(c) After lock picking.

Figure 5.4: Effects of lock picking on nested mutex bodies.

is based on the absence of $\pi$ functions for one or more lock variables at each mutex body lock node. Recall that the absence of $\pi$ functions for lock variables at lock nodes means that there are no concurrent threads trying to acquire that lock. This might make the lock operation unnecessary. These conditions are typically discovered using whole program analysis. For example, consider the program in Figure 5.4(a). The inner loop calls the function *sum_reduction* to update a global reduction variable. Since *sum_reduction* is a generic reduction function, it locks the variable before doing the reduction. However, as a result of inlining, reduction lock $S$ is no longer necessary because the reduction is always protected by lock $R$ (Figure 5.4(b)). When function *sum_reduction* is inlined, the use of lock $R$ at the lock node of the mutex body for $S$ becomes a protected use and its $\pi$ function can be removed (Novillo et al. 1998) (Figure 5.4(b)).

**Lemma 5.1 (Nested mutex structures)** Let $L = \{L_1, L_2, \ldots, L_m\}$ be the set of lock variables used in the program. Let $M_{L_j}$ be the mutex structure for lock variable $L_j$. If all the lock nodes in *every* mutex body of $M_{L_j}$ are lock-protected by the same lock variable $L_i$, then the lock and unlock nodes for mutex bodies in $M_{L_j}$ are unnecessary and can be removed. In this case, we say that mutex structure $M_{L_j}$ is *nested* inside mutex structure $M_{L_i}$. □

Proof Since all the lock nodes in all the mutex bodies in $M_{L_j}$ are lock-protected by the same lock variable $L_i$, all the `lock` operations on $L_j$ are serialized by lock $L_i$. Therefore, they are unnecessary because they are always guaranteed to succeed. Consequently, all the lock and unlock nodes for $L_j$ can be safely removed. ■

The second opportunity to lock-pick mutex bodies is when a particular mutex body cannot execute concurrently with any other mutex body of its same mutex structure. If this happens, we say that the mutex body is *non-conflicting*. Typically, a mutex body will be non-conflicting when it appears in sequential sections of a parallel program or if the program itself is sequential. Non-conflicting mutex bodies can also be discovered if all the mutex bodies in the same mutex structure are totally ordered by some other synchronization mechanism (e.g., `set/wait`, `barriers`, `coend` nodes). All the sequential programs described in Section 6.2 had their locks picked because

**Algorithm 5.2** Lock-picking.

INPUT: A CCFG in CSSAME form
OUTPUT: The graph with unnecessary lock and unlock operations removed

```
repeat
  /* First phase. Find nested mutex bodies. */
  foreach lock variable L_i do
    foreach mutex body B_{L_i}(N) ∈ M_{L_i} do
      foreach lock variable L_j do
        nested ← TRUE
        foreach node n ∈ N do
          if n contains a π function for L_j then
            nested ← FALSE
          end if
        end for
        if nested then
          Protectors(N) ← L_j
        end if
      end for
    end for
    if ⋂_N Protectors(N) ≠ ∅ then
      remove all lock and unlock nodes for mutex bodies in M_{L_i}
      update CSSAME information for L_i
    end if
  end for
  /* Second phase. Find non-conflicting mutex bodies. */
  foreach lock variable L_i do
    foreach mutex body B_{L_i}(N) ∈ M_{L_i} do
      hasConflicts ← FALSE
      foreach node n ∈ N do
        if n contains a π function for L_i then
          hasConflicts ← TRUE
        end if
      end for
      if not hasConflicts then
        remove all lock and unlock nodes for B_{L_i}(n)
        update CSSAME information for L_i
      end if
    end for
  end for
until no more changes have been made
```

they had no conflicts.

**Lemma 5.2 (Non-conflicting mutex bodies)** Let $M_L$ be the mutex structure for lock variable $L$. Let $B_L(N)$ be a mutex body in $M_L$. If no lock node $n \in N$ contains a $\pi$ function for $L$ then the `lock` and `unlock` operations for mutex body $B_L(N)$ are unnecessary and can be removed. □

PROOF If no lock node $n \in N$ contains a $\pi$ function for $L$ then no definition for $L$ comes from other concurrent threads. Since lock variables are defined at `lock(L)` nodes, this means that no other `lock(L)` node can execute concurrently with the lock nodes of $B_L(N)$. Therefore, the mutex body $B_L(N)$

is not necessary because all its lock nodes are guaranteed to acquire $L$ every time it executes. ∎

The conditions for lock picking given by these two lemmas have subtle differences that are worth noting. The conditions for Lemma 5.2 are only required to be met by a single mutex body. In contrast, Lemma 5.1 needs to check *all* the mutex bodies in the same mutex structure. It is not enough for one mutex body to be nested inside another. The whole mutex structure must be nested inside the *same* lock. Otherwise, the transformation cannot be done.

## 5.4 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 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.

Lock-Independent Code Motion (LICM) is a code motion technique that attempts to minimize the amount of code executed inside a mutex body. This optimization differs from lock picking in that it does not target the `lock` operations directly. Rather, it analyzes the mutex body itself to find code that can be moved outside. If at the end of the transformation a mutex body only contains unlock nodes, then the `lock` and `unlock` instructions are removed.

**Definition 5.1 (Lock-independence)** An expression $E$ inside a mutex body $B_L$ is lock-independent with respect to $L$ if moving $E$ outside $B_L$ does not change the meaning of the program. Similarly, a statement (or group of statements) $S$ is lock independent with respect to $L$ if all the expressions and definitions in $S$ are lock-independent. A flowgraph node $n$ is lock independent if all its statements are lock-independent. □

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

The concept of lock-independence is similar to the concept of loop-invariant code for standard loop optimization techniques (Aho et al. 1986). However, the conditions that make code to be lock-independent are different from those that make it loop invariant. 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.

### 5.4.1 Moving Lock-Independent Statements

Lock-independence is a necessary condition for moving a statement outside the mutex body, but it is not sufficient. The sufficient condition is that after the motion, the statement should preserve all its original control and data dependencies. For instance, if the statement is inside a loop it cannot be moved out unless it is also loop invariant. This section develops an algorithm to detect and move lock-independent statements outside mutex bodies. Sections 5.4.2 extends this to control structures and 5.4.3 deals with lock-independent expressions.

#### Moving Statements to Premutex Nodes

Given a lock-independent statement $s$ inside a mutex body $B_L(N)$, LICM will attempt to move $s$ to premutex or postmutex nodes for $B_L(N)$. This section describes the conditions required when attempting to move $s$ to premutex nodes for $B_L(N)$. The selection of `lock` nodes to receive statement $s$ in their premutex node is done satisfying the following conditions:

**Protection.** Candidate `lock` nodes are initially selected among all the `lock` nodes in $N$ that reach the node containing $s$ (denoted $node(s)$). For instance, consider the program in Figure

```
A = 0;
cobegin
  T0: begin
    x = 1;
    y = 0;
    done = 0;
    lock(L);
    while (!done) {
      y = y + 3;
      A = A + x;
      unlock(L);
      x = x + 1;
      if (x > 0) {
        lock(L);
        done = 1;
        x = x - A;
      } else {
        lock(L);
        A = A * x;
        x = x + 5;
      }
      y = y - 2;
    }
    if (A < x) {
      A = A + x;
      unlock(L);
      x -= 3;
    } else {
      A = A - x;
      unlock(L);
    }
    print(A, x, y);
  end

  T1: begin
    lock(L);
    A += f();
    unlock(L);
  end
coend
```

(a) Original program.

```
    A = 0;
    cobegin
      T0: begin
        x = 1;
        y = 0;
        done = 0;
        lock(L);
        while (!done) {
          A = A + x;
          unlock(L);
          x = x + 1;
          if (x > 0) {
⇒            y = y + 3;
⇒            done = 1;
            lock(L);
            x = x - A;
          } else {
⇒            y = y + 3;
            lock(L);
            A = A * x;
            x = x + 5;
          }
          y = y - 2;
        }
        if (A < x) {
          A = A + x;
          unlock(L);
          x -= 3;
        } else {
          A = A - x;
          unlock(L);
        }
        print(A, x, y);
      end

      T1: begin
        lock(L);
        A += f();
        unlock(L);
      end
    coend
```

(b) After LICMS.

Figure 5.5: Moving lock-independent statements. Moved statements are marked with arrows (⇒).

5.5(a). Thread $T_0$ contains one mutex body $B_L(\{7, 14, 18\}) = \{8, 9, 10, 11, 15, 16, 19, 20, 21, 22, 23, 24, 25, 28, 29\}$[1]. Statement $A = A{+}x$ at line 10 is reached by the `lock` nodes at lines 7, 14 and 18. However, statement $x = x + 5$ at line 20 is only reached by the `lock` node at line 18. This condition provides an initial set of candidate `lock` nodes called $\mathit{protectors}(s)$.

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

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

Besides having multiple premutex nodes that could receive $s$, a mutex body could have multiple combinations of receiver nodes for $s$. For instance, in the program fragment of Figure 5.5(a), lock-independent statement $s : y = y + 3$ at line 9 is reached by `lock` nodes 7, 14 and 18. For the purpose of this discussion we disregard other considerations that might prevent moving $s$ outside the mutex body (e.g., data dependencies). Notice that moving $s$ to all three premutex nodes is not a valid option because this creates duplicate instances of $s$ on a single control path. There are two sets of receiver nodes for $s$ in this program, namely $\{7\}$ and $\{14, 18\}$. Further analysis will determine which of these receiver sets is the better choice.

[1] For simplicity we are assuming that each line corresponds to a node in the CCFG.

Algorithm 5.3 computes all the different sets of `lock` nodes that may receive a lock-independent statement $s$ in their premutex nodes. Basically, the algorithm computes reachability sets among the nodes in $\mathit{protectors}(s)$. The set $\mathit{protectors}(s)$ is partitioned into $k$ partitions $P_1, P_2, \ldots P_k$. Nodes in each partition $P_j$ cannot reach each other but put together they reach or are reached by every other node in $\mathit{protectors}(s)$. These partitions are the sets of `lock` nodes that can receive a copy of $s$ in their premutex nodes.

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

1. No variable defined by $s$ may be used or defined along any path from $\mathit{node}(s)$ to every node in $P_j$.
2. No variable used by $s$ may be defined along any path from $\mathit{node}(s)$ to every node in $P_j$.

These two restrictions are used to prune the set of receiver nodes computed in Algorithm 5.3. Notice that since the program is in CSSAME form, $\phi$ functions are also considered definitions and uses for a variable.

In the example program of Figure 5.5(a) the receiver node for statement $x = x + 5$ at line 20 is node 18, which cannot receive it because $x$ is used at line 19. Statement $y = y + 3$ has two sets of receiver nodes: $\{7\}$ and $\{14, 18\}$. Node 7 cannot be used because of the $\phi$ function for $y$ at the head of the `while` loop. However, both nodes 14 and 18 could receive a copy of the statement.

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

**Algorithm 5.3** Compute candidate premutex nodes (*receivers*).

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

*protectors*$(s) \leftarrow$ set of `lock` nodes that reach $s$.

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

**return** *receivers* $\leftarrow P(1), P(2), \ldots, P(k-1)$

**Theorem 5.2 (Hoistable statements)** Let $s$ be a lock-independent statement $s$ inside a mutex body $B_L(N)$. Let *protectors*$(s)$ be a set of `lock` nodes in $N$ such that:

1. $\forall n_i \in$ *protectors*$(s)$ : node $n_i$ reaches *node*$(s)$,
2. there exist $k$ partitions $P : P_1, P_2, \ldots, P_k$ $(k \geq 1)$ of the set *protectors*$(s)$ computed as per Algorithm 5.3, and
3. there exists a partition $P_j \in P$ for which (a) no variable defined by $s$ is defined nor used in any path between *node*$(s)$ and nodes in $P_j$, and (b) no variable used by $s$ is defined in any path between *node*$(s)$ and nodes in $P_j$.

If these conditions hold for at least one partition $P_j$ then it is possible to move $s$ to the premutex nodes for the `lock` nodes in $P_j$. □

Proof Since *node*$(s)$ is reached by every node $n_i \in$ *protectors*$(s)$, there exists a path between $n_i$ and *node*$(s)$. Let $P_j$ be a set of nodes that complies with the three conditions in the theorem. The nodes in $P_j$ have the following properties:

1. $\forall n_i, n_k \in P_j$ such that $n_i \neq n_k$, there is no control path between $n_i$ and $n_k$. This is immediate from the way the algorithm selects the nodes (lines 9-10 of Algorithm 5.3).

2. $\forall n_i \in \mathit{protectors}(s)$ : if $n_i \notin P_j$ then $\exists n_k \in P_j$ such that there is a path between $n_i$ and $n_k$. Suppose that there is a node $n_i \in \mathit{protectors}(s)$ that cannot be reached by any node in $P_j$ then the algorithm would have placed $n_i$ in $P_j$, which is a contradiction.

The previous two conditions guarantee that if $s$ is removed from $\mathit{node}(s)$ and replicated to every node in $P_j$ then one and only one instance of $s$ will still be available on paths leading to or from nodes in $\mathit{protectors}(s)$. Finally, let $D_s$ be the set of variables defined in $s$. Since no path between $\mathit{node}(s)$ and the nodes in $P_j$ defines or uses a variable in $D_s$, moving $s$ will not alter data dependencies for $s$. Similarly, let $U_s$ be the set of variables used in $s$. Since no path between $\mathit{node}(s)$ and $n_i$ defines defines variables in $U_s$, it is safe to move $s$. ■

### Moving Statements to Postmutex Nodes

The LICM transformation may also move statements to postmutex nodes of a mutex body $B_L(N)$. The analysis for postmutex nodes is similar to the previous case. The conditions are essentially the reverse of the conditions required for premutex nodes.

**Protection.** Unlock node $x_i$ must be reached by the same `lock` nodes that reach statement $s$. This guarantees that there exists a control path between $\mathit{node}(s)$ to $x_i$. This condition provides an initial set of `unlock` nodes to consider as candidates. In the example program in Figure 5.5(a), the statement $y = y + 3$ at line 9 is reached by `lock` nodes 7, 14 and 18 which also reach `unlock` nodes 11, 26 and 30.

**Reachability.** Algorithm 5.4 computes all the different sets of `unlock` nodes that may receive a lock-independent statement $s$ in their postmutex nodes. The algorithm performs the same reachability analysis done by Algorithm 5.3. The set $\mathit{releasers}(s)$ contains all the `unlock` nodes reached by the same `lock` nodes that reach $s$. The set *releasers*$(s)$ is partitioned into $k$ partitions $X_1, X_2, \ldots X_k$. Nodes in each partition $X_j$ cannot reach each other but put together they reach or are reached by every other node in *releasers*$(s)$. These partitions are the sets of `unlock` nodes that can receive a copy of $s$ in their postmutex nodes.

**Data dependencies.** The same requirements needed for premutex nodes are necessary for postmutex nodes. If any variable defined by $s$ is defined or used in any path from $s$ to a node in *releasers*$(s)$ then $s$ may not be moved. Similarly, if any variable used by $s$ is defined in any path from $s$ to a node in *releasers*$(s)$ then $s$ may not be moved.

**Algorithm 5.4** Compute candidate postmutex nodes (*releasers*).

INPUT: A mutex body $B_L(N)$ and a lock-independent statement $s$.
OUTPUT: A list of releaser sets. Each releaser set $X_i$ contains the `unlock` nodes whose postmutex nodes may receive $s$.

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

**Theorem 5.3 (Downward-movable statements)** Let $s$ be a lock-independent statement $s$ inside a mutex body $B_L(N)$. Let *releasers*$(s)$ be a set of `unlock` nodes in $B_L$ such that:

1. $\forall x_i \in \mathit{releasers}(s)$ : node $x_i$ is reached by a node in $\mathit{protectors}(s)$,

2. there exist $k$ subsets $X : X_1, X_2, \ldots, X_k$ ($k \geq 1$) of the set $\mathit{releasers}(s)$ computed as per Algorithm 5.4, and

3. there exists a partition $X_j \in X$ for which (a) no variable defined by $s$ is defined nor used in any path between $\mathit{node}(s)$ and nodes in $X_j$, and (b) no variable used by $s$ is defined in any path between $\mathit{node}(s)$ and nodes in $X_j$.

If these conditions hold for at least one partition $X_j$ then it is possible to move $s$ to the postmutex nodes for the `unlock` nodes in $X_j$. □

PROOF Similar to the proof for Theorem 5.2. ■

### LICM for Statements (LICMS)

Theorems 5.2 and 5.3 are used as the basis for the algorithm to move statements outside mutex bodies (Algorithm 5.5). Notice that even though we refer to *hoistable* statements for statements that can be moved to a premutex node, the movement is not necessarily made against the flow of control. The name was chosen because that is what happens in the most general case. Similarly, *downward-movable* statements may be moved up.

The LICMS algorithm scans all the mutex bodies in the program looking for lock-independent statements to move outside the mutex body. Each lock-independent statement $s$ is checked against the conditions described previously. Lines $8 - 15$ in Algorithm 5.5 determine the sets of premutex receivers for $s$. The initial set of candidates computed by Algorithm 5.3 checks every lock node in a mutex body against each other looking for paths between them. If $mb$ is the number of mutex bodies in the program, this can be accomplished in $O(mb^2)$ time. To check data dependencies each statement has to be compared with all the statements in paths to each premutex node (lines $9 - 15$). Given that there may be up to $mb$ premutex nodes, data dependencies can be checked in $O(mb \times |S|^2)$, where $S$ is the set of statements in the program. This yields a total time complexity for lines $8 - 15$ of $O(mb^2 + mb \times |S|^2)$. Similarly, lines $16 - 24$ compute sets of postmutex receivers in time $O(mb^2 + mb \times |S|^2)$.

Notice that it might be possible that a statement can be moved to both the premutex and the postmutex nodes. In that case a cost model should determine which node is more 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 premutex nodes will not be affected by it because they execute before acquisition of the lock. However, statements moved to the postmutex node will be delayed if there is contention because they execute after the lock has been released. Therefore, when a statement can be moved to both the premutex and postmutex nodes, the premutex node is selected.

When more than one set of premutex or postmutex nodes can receive a statement $s$ a cost model should be use to select the more profitable target. Although not addressed in this document, cost models may include simple factors like checking that statements are not moved into loops or even delaying all the hoisting decisions until the algorithm has finished analyzing all the statements in a single mutex body.

Finally, if the mutex body is empty at the end of the transformation, the `lock` and `unlock` nodes are removed (lines $36-39$). The total time complexity for the LICMS algorithm is then $O(m \times mb \times (mb^2 + mb \times |S|^2))$. In general, we expect the cost to be dominated by $|S|$ because $m$ (number of lock variables) and $mb$ (number of mutex bodies in the program) will be relatively small compared to $|S|$. The effects of LICMS on the program in Figure 5.5(a) are shown in Figure 5.5(b). Notice that the statement $y = y + 3$ at line 9 in Figure 5.5(a) as been replicated into lines 13 and 18 in the transformed program of Figure 5.5(b). It is necessary to replicate the statement, otherwise the transformed program will not compute the same value of $y$ than the original one.

### 5.4.2 LICM for Control Structures

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

**Algorithm 5.5** Lock-Independent Code Motion for Statements (LICMS).

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

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

```
foreach lock variable L_i do
  foreach mutex body B_{L_i}(N) ∈ MutexStruct(L_i) do
    n_i ← node(L_i)
    foreach lock-independent statement s reached by n_i do
      D_s ← variables defined by s
      U_s ← variables used by s

      /* Determine which premutex nodes can receive s. */
      P ← receivers of s at premutex nodes (Algorithm 5.3)
      foreach P_i ∈ P do
        foreach node n ∈ P_i do
          if (any path between n and node(s) defines or uses a variable in D_s)
          or (any path between n and node(s) defines a variable in U_s) then
            remove P_i from P
          end if
        end for
      end for

      /* Determine which postmutex nodes can receive s. */
      X ← receivers of s at postmutex nodes (Algorithm 5.4)
      foreach X_i ∈ X do
        foreach node x ∈ X_i do
          if (any path between x and node(s) defines or uses a variable in D_s)
          or (any path between x and node(s) defines a variable in U_s) then
            remove X_i from X
          end if
        end for
      end for

      /* Sets P and X contain sets of premutex and postmutex nodes that can receive s. */
      if P ≠ ∅ then
        select one P_i ∈ P (cost model or random)
        remove s from its original location
        replicate s to each node n ∈ P_i
      else if X ≠ ∅ then
        select one X_i ∈ X (cost model or random)
        remove s from its original location
        replicate s to each node x ∈ X_i
      end if
    end for

    /* Remove the mutex body if it is empty. */
    if B_{L_i}(N) = ∅ then
      remove all the lock and unlock nodes of B_{L_i}(N)
    end if
  end for
end for
```

Algorithm 5.5 can be used to hoist the structures outside mutex bodies.

Algorithm 5.6 looks for control structures that only contain lock-independent statements. Control structures are identified using standard interval analysis techniques (Aho et al. 1986). Basically, control structures 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 identified, sub-graphs inside a mutex body are scanned to determine if all their interior statements are lock-independent. If so, the variables defined and used by each statement are aggregated into the sets $D_H$ and $U_H$ for each sub-graph (lines $9-22$ in Algorithm 5.6). After all the sub-graphs in every mutex body of the program have been identified, Algorithm 5.5 is used to hoist them out of mutex bodies. The identification of lock-independent sub-graphs can be done in $O(m \times mb \times |S|)$ time. Where $m$ is the number of lock variables used in the program, $mb$ the number of mutex bodies and $S$ is the set of statements in the program.

**Algorithm 5.6** LICM for Control Structures (LICMT).

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 premutex and postmutex nodes

1: build sub-graphs for all control structures in the program
2: **foreach** lock variable $L_i$ **do**
3:   **foreach** mutex body $B_{L_i}(N) \in MutexStruct(L_i)$ **do**
4:     /* Build sub-graphs for all the control structures in the mutex body. */
5:     /* Find lock-independent sub-graphs. */
6:     **foreach** subgraph $H$ inside $B_{L_i}(N)$ **do**
7:       $D_H \leftarrow \emptyset$
8:       $U_H \leftarrow \emptyset$
9:       **foreach** statement $s$ in $H$ **do**
10:         **if** $s$ is **not** lock-independent **then**
11:           mark $H$ as lock-dependent (i.e., it cannot be moved)
12:           continue with next sub-graph
13:         **else**
14:           /* Add defines and uses made by $s$ to the sub-graph. */
15:           $D_H \leftarrow D_H \bigcup D_s$
16:           $U_H \leftarrow U_H \bigcup U_s$
17:         **end if**
18:       **end for**
19:       mark $H$ as lock-independent
20:     **end for**
21:   **end for**
22: **end for**
23: hoist lock-independent sub-graphs using Algorithm 5.5

### 5.4.3 LICM for Expressions

If hoisting statements or control structures outside mutex bodies is not possible, it may still be possible to consider moving lock-independent sub-expressions outside mutex bodies. This strategy is similar to moving statements (Algorithm 5.5) with the following differences:

1. Sub-expressions do not define variables. They only read variables or program constants.

2. If a sub-expression is moved from its original location, the computation performed by the expression must be stored in a temporary variable created by the compiler. The original expression is then replaced by the temporary variable. This is the same substitution performed by common sub-expression and partial redundancy elimination algorithms (Aho et al. 1986; Chow et al. 1997).

3. Contrary to the case with statements and control structures, expressions can only be moved against the flow of control. The reason is that the value computed by the expression needs to be available at the statement containing the original expression.

Algorithm 5.7 finds and removes lock-independent expressions from mutex bodies in the program. The process of gathering candidate expressions is similar to that of SSAPRE, an SSA based partial redundancy elimination algorithm (Chow et al. 1997). Mutex bodies are scanned for lock-independent first-order expressions, which are expressions that contain only one operator. Higher order expressions are handled by successive iterations of the algorithm.

Once lock-independent expressions are identified, the algorithm looks for suitable premutex or postmutex nodes to receive each expression. We observe that since expressions can only be hoisted up in the graph, it is not necessary to consider postmutex nodes when moving lock-independent expressions.

**Theorem 5.4 (Target nodes for lock-independent expressions)** Let $e$ be a lock-independent expression inside mutex body $B_L(N)$. If $e$ can be hoisted to a postmutex node of $B_L(N)$ there exists a premutex node of $B_L(N)$ that can also receive $e$. □

**Algorithm 5.7** Lock-Independent Code Motion for Expressions (LICME).

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

1: **repeat**
2:   **foreach** lock variable $L_i$ **do**
3:     **foreach** mutex body $B_{L_i}(N) \in M_{L_i}$ **do**
4:       $E \leftarrow E \bigcup$ set of lock-independent expressions in $B_{L_i}(N)$.
5:       **if** $E \neq \emptyset$ **then**
6:         **foreach** expression $E_j \in E$ **do**
7:           $P \leftarrow$ premutex receivers for $E_j$ (Algorithm 5.3)
8:           $candidates \leftarrow \emptyset$
9:           **foreach** $P_i \in P$ **do**
10:             **if** $\forall n \in P_i : (n\ DOM\ node(E_j))$ **or** $(node(E_j)\ PDOM\ n)$ **then**
11:               $candidates \leftarrow P_i$
12:               stop looking for candidates
13:             **end if**
14:           **end for**
15:           **if** $candidates \neq \emptyset$ **then**
16:             insert the statement $t_j = E_j$ in all the premutex nodes for `lock` nodes in *candidates*
17:           **end if**
18:         **end for**
19:       **end if**
20:     **end for**
21:   **end for**
22:   /* Replace hoisted expressions inside each mutex body. */
23:   **foreach** lock variable $L_i$ **do**
24:     **foreach** mutex body $B_{L_i}(N) \in M_{L_i}$ **do**
25:       replace hoisted expressions in $B_{L_i}(N)$ with their corresponding temporaries
26:     **end for**
27:   **end for**
28: **until** no more changes have been made

PROOF Let $x$ be an `unlock` node in $B_L(N)$ such that $postmutex(x)$ can receive $e$. Since $e$ can only be moved against the flow of control, there exists a control path from $x$ to $node(e)$. Furthermore, since $e$ is inside the mutex body, $node(e)$ must be reached by some `lock` node $n \in N$ such that *every* path from $x$ to $node(e)$ goes through $n$. Therefore, if $e$ can be placed in $postmutex(x)$ it can also be moved to $premutex(n)$. ■

We use the previous result to reduce the number of candidate nodes to be considered when moving lock-independent expressions. Only `lock` nodes are considered by the algorithm. Furthermore, the candidate `lock` must dominate or be post-dominated by the node holding the expression (lines $7 - 13$ in Algorithm 5.7).

The acceptable receiver sets are stored in the set *candidates*. Using a similar reasoning to Theorem 5.4 it can be shown that in this case, the algorithm for computing receiver premutex nodes (Algorithm 5.3) will find none or exactly one set of `lock` nodes that can receive the expression in their premutex nodes.

Figure 5.6 shows an example program before and after running the LICM algorithm. When LICM is applied to the program in Figure 5.6(a), the first phase of the algorithm moves the statement at line 9 and the assignment $j = 0$ to the premutex node. The statement at line 13 is sunk to the postmutex node resulting in the equivalent program in Figure 5.6(b). There is still some lock-independent code in the mutex body, namely the expressions $j < M$ at line 11, the statement $j{+}{+}$ at line 11 and the expression $y[j] + sqrt(a) * sqrt(b)$ at line 12. The only hoistable expression is $sqrt(a) * sqrt(b)$ because it is the only expression with all its reaching definitions outside the mutex body. Note that a loop-invariance transformation would have detected this expression and hoisted it out of the loop. LICM goes a step further and hoists the expression outside the mutex body.

### 5.4.4 Putting it All Together: Lock-Independent Code Motion (LICM)

The individual algorithms discussed in previous sections can be combined into a single LICM algorithm (Algorithm 5.8). There are four main phases to the algorithm. The first phase looks for mutex bodies that have nothing but lock-independent nodes. These are the simplest cases. If all the nodes in a mutex body are lock-independent, then the `lock` operations at the lock nodes and the `unlock` operations in the body can be removed. The next three phases move interior lock-independent statements, control structures and expressions outside the mutex bodies in the program (Algorithms 5.5, 5.6 and 5.7). We show the effect of the LICM transformation in several explicitly parallel programs in Chapter 6.

## 5.5 Mutex Body Localization

In this section we discuss a transformation technique that may enhance the opportunities for further optimization of the program. Consider a mutex body

```
double X[]; /* shared */

parloop (i, 0, N) {
  double a, b; /* local */
  double y[]; /* local */

  ...
  lock(L);
  b = a * sin(a);
  for (j = 0; j < M; j++) {
    X[j] = y[j] + sqrt(a) * sqrt(b);
  }
  a = y[i];
  unlock(L);
  ...
}
```

(a) Program before LICM.

```
double X[]; /* shared */

parloop (i, 0, N) {
  double a, b; /* local */
  double y[]; /* local */

  ...
  b = a * sin(a);
  j = 0;
  lock(L);
  for (; j < M; j++) {
    X[j] = y[j] + sqrt(a) * sqrt(b);
  }
  unlock(L);
  a = y[i];
  ...
}
```

(b) After LICM on statements.

```
double X[]; /* shared */

parloop (i, 0, N) {
  double a, b; /* local */
  double y[]; /* local */

  ...
  b = a * sin(a);
  j = 0;
  t1 = sqrt(a) * sqrt(b);
  lock(L);
  for (; j < M; j++) {
    X[j] = y[j] + t1;
  }
  unlock(L);
  a = y[i];
  ...
}
```

(c) After LICM on expressions.

Figure 5.6: Effects of lock-independent code motion (LICM).

**Algorithm 5.8** Lock-Independent Code Motion (LICM).

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

```
/* First phase. Try to remove lock and unlock nodes for mutex bodies with nothing but LI nodes. */
foreach lock variable L_i do
  foreach mutex body B_{L_i}(N) do
    if all the nodes a ∈ B_{L_i}(N) are lock independent then
      remove all lock and unlock nodes for B_{L_i}(N)
    end if
  end for
end for
/* Second phase. Move whole control structures out. */
perform LICM on structures (Algorithm 5.6)
/* Third phase. Move individual statements out. */
perform LICM on statements (Algorithm 5.5)
/* Fourth phase. Try to move expressions. */
perform LICM on expressions (Algorithm 5.7)
```

$B_L$ that modifies a shared variable $V$ (Figure 5.7(a)). With the exception of the definition reaching the unlock node of $B_L$, all the modifications done to $V$ inside the mutex body can only be observed by the thread.

Given these conditions, it is possible to create a local copy of $V$ and replace all the references to $V$ inside the mutex body to references to the local copy (Figure 5.7(b)). We call this transformation *mutex body localization* (MBL). It is the dual technique to LICM. While LICM looks for lock-independent code, MBL creates lock-independent code by modifying the left-hand side of statements. The basic transformation is straightforward:

1. At the start of the mutex body a local copy of the shared variable is created if there is at least one use for the variable with reaching definitions outside the mutex body.

2. At the mutex body exits, the shared copy is updated from the local copy of the variable if at least one internal definition of the variable reaches that particular unlock node.

3. All the interior references to the shared variable are modified so that they reference the local copy.

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

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

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

(a) A mutex body before localization.

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

(b) After localization.

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

(c) After reduction recognition.

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

(d) After LICM.

Figure 5.7: Applications of mutex body localization.

Algorithm 5.10 makes local copies of a variable $a$ inside a mutex body $B_L(N)$ if the variable can be localized. To determine whether the variable $a$ can be localized it calls Algorithm 5.9 (a subroutine of Algorithm 5.10) which returns TRUE if $a$ can be localized inside mutex body $B_L(N)$. The localization algorithm relies on two data structures that can be built during the $\pi$ rewriting phase of the CSSAME algorithm (Algorithm 4.5):

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

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

Algorithm 5.10 starts by checking whether the variable can be localized (lines $1-4$). It then checks where the local copies are needed. If there are upward-exposed uses of $a$, a copy is needed at the start of the mutex body (lines $5-16$). If there are definitions of $a$ reaching an exit node, the shared copy of $a$ must be updated before exiting the mutex body (lines $17-29$). The final phase of the algorithm updates the interior references to $a$ to be references to $p\_a$ (lines $30-34$). After this phase, the CSSAME form for the program has been altered and it should be updated. The simplest way to do this is to run the CSSAME algorithm again (Algorithm 4.7). 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 $\pi$ function. The argument list to this $\pi$ function is the union of all the arguments to all the $\pi$ functions for $a$ inside the mutex body. Notice that the $\pi$

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 $\pi$ 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 $\phi$ functions for $a$ must be converted into $\phi$ functions for $p\_a$.

4. If the shared copy is updated at the end of the mutex body, the statement $a = p\_a$ contains a use of $p\_a$ whose control reaching definition should be the definition of $p\_a$ reaching the exit node $x$.

**Algorithm 5.9** Localization test (*localizable*).

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

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

The MBL 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 create more lock-independent code. For instance, if a thread contains read-only references to a variable $V$, localizing $V$ will make those reads into lock-independent operations which in turn might make the whole statement lock-independent. Consider the sample program in Figure 5.7(a). After localization (Figure 5.7(b)), most statements

**Algorithm 5.10** Mutex body localization.

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

OUTPUT: $B_L(N)$ with variable $a$ localized

```
/* Check if a can be localized (Algorithm 5.9) */
if not localizable(a, B_L(N)) then
    return
end if
/* Check for upward-exposed uses of a. Since the program is in CSSAME form, */
/* upward-exposed uses have already been computed (Algorithm 4.5). If there are */
/* upward-exposed uses of a then we need to make a local copy of a at the start of B_L(N). */
needEntryCopy ← FALSE
foreach use u ∈ exposedUses(N) do
    if u is a use of a then
        needEntryCopy ← TRUE
    end if
end for
if needEntryCopy then
    insert the statement p_a = a at the start of the mutex body
end if
/* Check if any definition of a reaches the exit nodes of B_L(N). */
/* Since the program is in CSSAME form, the definitions that reach the exit nodes X */
/* have already been computed (Algorithm 4.5). If a definition */
/* of a reaches x, we need to make a copy of a before leaving the mutex body. */
needExitCopy ← FALSE
foreach definition d ∈ reachingDefs(X) do
    if d is a definition of a then
        needExitCopy ← TRUE
    end if
end for
if needExitCopy then
    insert the statement a = p_a at the exit nodes of the mutex body
end if
/* Update references to a inside the mutex body to reference */
/* the local version p_a instead of the shared version a. */
foreach reference to a inside B_L(N) do
    replace a with p_a
end for
update CSSAME information for all references to p_a inside B_L(N)
```

inside the mutex body for $L$ are lock-independent. However, none can be moved outside because of the read and write operations to the shared variable $V$ at the fringes of the mutex body. If the compiler incorporates a reduction recognition pass, it is possible to do the reduction locally and only update $V$ at the end (Figure 5.7(c)). Now all the lock-independent code in the mutex body can be moved to the premutex node resulting in the equivalent program in Figure 5.7(d). As we will discuss in Chapter 6 this is a common transformation performed manually by programmers. Using these techniques, it is possible to make this transformation automatically in the compiler.

### 5.5.1 Single Writer, Multiple Readers Lock Picking

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 $V$),
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),
3. within the concurrent threads (i.e., the writer $T_w$ and all the readers), variable $V$ is only accessed inside critical sections of the code, and
4. the underlying memory model is strongly consistent.

Under these circumstances it is possible to localize the references to $V$ in $T_w$ so that atomicity can be maintained without requiring locks. For example, consider the program in Figure 5.8(a). Thread $T_0$ computes a value for $V$, checks a bound and updates $V$ if necessary (assume that global variables $X$ and $Y$ have no conflicts). Both threads $T_1$ and $T_2$ read $V$ but never modify it. The synchronization on $V$ is necessary to prevent threads $T_1$ and $T_2$ from reading intermediate values of $V$ while $T_0$ computes. Suppose that we localize variable $V$ inside $T_0$ to obtain the equivalent program in 5.8(b). Since $X$ and $Y$ contain no conflicts and the references to $V$ 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 5.8(c). Finally, since thread $T_0$ writes to $V$ only once, the locks are not really necessary and can be removed to obtain the equivalent program in in Figure 5.8(d).

## 5.6 Summary

In this chapter we used the CSSAME framework to develop two types of optimizing transformations: the adaptation of sequential techniques to work on

```
X = ...
Y = ...
cobegin
  T0: begin
    ...
    lock(L);
    a = 0;
    while (a <= X) {
      a = a + Y;
    }

    unlock(L);
  end

  T1: begin
    lock(L);
    ... = a;
    unlock(L);
  end

  T2: begin
    lock(L);
    ... = a;
    unlock(L);
  end
coend
```

(a) Original program.

```
X = ...
Y = ...
cobegin
  T0: begin
    ...
    lock(L);
    p_a = 0;
    while (p_a <= X) {
      p_a = p_a + Y;
    }
    a = p_a;
    unlock(L);
  end

  T1: begin
    lock(L);
    ... = a;
    unlock(L);
  end

  T2: begin
    lock(L);
    ... = a;
    unlock(L);
  end
coend
```

(b) After localization.

```
X = ...
Y = ...
cobegin
  T0: begin
    ...
    p_a = 0;
    while (p_a <= X) {
      p_a = p_a + Y;
    }
    lock(L);
    a = p_a;
    unlock(L);
  end

  T1: begin
    lock(L);
    ... = a;
    unlock(L);
  end

  T2: begin
    lock(L);
    ... = a;
    unlock(L);
  end
coend
```

(c) After LICM.

```
X = ...
Y = ...
cobegin
  T0: begin
    ...
    p_a = 0;
    while (p_a <= X) {
      p_a = p_a + Y;
    }

    a = p_a;

  end

  T1: begin

    ... = a;

  end

  T2: begin

    ... = a;

  end
coend
```

(d) After relaxing lock independence.

Figure 5.8: Effects of MBL in the presence of single-writer, multiple-readers.

explicitly parallel programs and the direct optimization of the synchronization structure of a parallel program. To our knowledge the techniques presented in this chapter are the first to address the problem of optimizing mutual exclusion structures in an explicitly parallel program.

These transformations will benefit explicitly parallel programs that use mutex synchronization frequently. In particular, programs that make use of thread-safe libraries (e.g., multi-threaded Java applications) may contain superfluous mutex synchronization that slow down the program unnecessarily. In this context we observed that these techniques can have a significant impact on performance. Even sequential programs can benefit from these transformations. In the following chapter we study the effectiveness of these techniques in several C and Java applications.

# Chapter 6

# Results

The techniques developed in this thesis are the first step towards a general optimizing compiler for explicitly parallel programs. We have implemented many of the analysis and optimization algorithms presented in this thesis into a compiler for the C language. All the example program fragments described in previous chapters have been analyzed and optimized by our compiler. We have also been able to perform experiments to demonstrate the potential for some of these techniques in complete programs.

We studied two main types of applications: those in which the user has little control over synchronization structures in the program and those in which the user has complete control over all the synchronization used in the program.

Applications in the first group are developed in languages that expose most of the synchronization and parallelism details. We have selected some applications from the SPLASH suite of shared-memory parallel programs (Singh et al. 1992) and applications bundled with the TreadMarks DSM system (Keleher et al. 1994). These applications represent code developed by expert programmers who are very conscious about the performance implications of synchronization operations. The synchronization structures found in these applications have been optimized manually by the programmer. As a consequence we did not expect to find many opportunities for optimization in the context of the techniques developed in this thesis. However, we did find that some of the manual modifications made by the programmer could have been performed automatically using our techniques.

The second group consists of applications typically developed in programming environments that produce generic skeleton code and systems that provide thread-safe libraries. 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 (Bacon et al. 1998). 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 benefit these applications. Similar benefits are obtained in parallel programs generated via high-level programming environments. These tools must generate conservatively correct code, and are typically based on code skeletons that, because of their generality, must contain over-constrained synchronization. Similar to the previous case, machine generated code must be overly conservative for generality and safety.

## 6.1 Implementation

Many of the algorithms discussed in previous sections have been implemented[1] in a prototype compiler for the C language using the SUIF compiler system (Hall et al. 1996). 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 specially named 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

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

parallel and modifies the index variable to be the thread id. Parallel loops are recognized using a similar technique. A `parloop` is a `for` loop with a specially named index variable. Since SUIF directly supports `parloop` style parallelism all the compiler has to do is mark selected `for` loops as parallel loops.

Once the program has been parsed by the SUIF front-end, the compiler creates the corresponding CCFG and its CSSAME form. We do not transform the input program into SSA form. Instead we use factored use-def chains (Wolfe 1996) in the flowgraph and display the source code annotated with the appropriate $\pi$ and $\phi$ functions (variables are not renamed but referenced using line number information in the corresponding $\pi$ or $\phi$ functions). The CCFG implementation is an extension of the sequential Control Flow Graph library provided by Machine SUIF (Holloway and Young 1997). The CCFG can be displayed using a variety of graph visualization systems. The flow graphs in this thesis were generated by the compiler and laid out using the GraphViz system (North and Koutsofios 1994). The CSSAME form for the program can also be displayed as an option. Finally, the mutual exclusion validation techniques discussed in Section 3.3.2 are implemented as compile-time warnings to the user.

A basic form of inter-procedural analysis (IPA) information is gathered by the current implementation. At each procedure call, shared variables referenced and mutex bodies defined by the called procedure are propagated to the call site. This allows the conflict and synchronization analyzer to treat function calls almost as if they were inlined code. Finally, we have implemented partial support for reductions based on the SUIF reduction recognizer. Currently, the compiler is limited to reductions inside `for` loops.

## 6.2 Experimental Results

Synchronization overhead is sometimes caused by an expensive implementation of `lock` and `unlock` operations. To address this problem, several techniques have been proposed to implement more efficient locking primitives (Bacon et al. 1998; Mellor-Crummey and Scott 1991; Unrau et al. 1994). The techniques for eliminating superfluous synchronization operations developed in this thesis can complement the benefits of using an efficient locking mechanism.

There is another source of overhead that even the most efficient implementation cannot alleviate: contention. Lock contention occurs when the demand for a particular lock variable is so high that threads spend a significant amount of time waiting for other threads to release the lock. In the following sections we demonstrate the effects of the techniques developed in this thesis on several programs. Section 6.2.1 describes two applications from the SPLASH suite. Section 6.2.2 studies some parallel and sequential Java programs.

Note that at the time of this writing, the compiler is not yet ready to tackle all the programs described in this section. In the current implementation, alias analysis is limited to simple pointer aliasing: the compiler only detects aliases for pointers that explicitly take the address of a shared variable. The compiler also lacks array analysis; it treats arrays as atomic memory references. The Omega library (Pugh and Wonnacott 1992) could be used to perform array section analysis. Alternatively, the array SSA form proposed by Collard (Collard 1999) could be used. This work is beyond the scope of the thesis.

Because of these limitations we simplified the input program for some of these applications to help the current implementation analyze and optimize the code. The modifications included replacing the original thread creation code with parallel loops and/or `cobegin/coend` structures, inlining some functions to circumvent limitations during synchronization analysis and substituting arrays of locks by single scalar lock variables. Once the compiler analyzed and optimized the simplified version, we made the same modifications to the original programs. This process was applied to the applications in Sections Sections 6.2.1 and 6.2.3.

The framework developed in this thesis cannot be directly applied to Java because Java has a different high-level model for concurrency and synchronization. However, we believe that it is possible to adapt the techniques developed in this document to fit the Java model. As a preliminary feasibility study, we manually applied the transformation algorithms to a set of Java applications. The results of our experimentation are described in Section 6.2.2 where we describe the results and the potential performance benefits of adapting our transformations to Java.

### 6.2.1 SPLASH Applications

SPLASH (Stanford Parallel Applications for Shared-Memory) (Singh et al. 1992; Woo et al. 1995) is a benchmark suite for shared memory architectures designed as a case study to evaluate different issues in shared memory architectures. In the following sections we discuss our optimization techniques in the context of two SPLASH applications: Water and Ocean.

Some of the mutual exclusion synchronization structures used in these applications were manually optimized by the original developers. We will show that using the techniques described in this thesis, it would have been possible to obtain similar performance benefits without the added complexity of manually modifying the code.

#### Water

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

The computation of inter-molecular interactions is synchronized using one lock per molecule. The code fragment in Figure 6.1 shows the mutex bodies in the procedure `UPDATE_FORCES`. Each mutex body updates a shared three-dimensional array. The right hand side of each expression is lock-independent. After the LICM transformation, the mutex bodies in this procedure are converted to their equivalent versions shown in Figure 6.2 (for space reasons we only include the first mutex body, the modifications to the second mutex body are identical). The transformation hoisted the right-hand side of every assignment statement to the temporary variables $t_1, t_2, \ldots t_9$. Furthermore, the address computation needed to perform the array references are also lock-independent. Therefore, the compiler was able to move the assignments to variables $suif\_tmp_{19}, suif\_tmp_{21}, \ldots suif\_tmp_{35}$

```
UPDATE_FORCES(DEST, mol, comp, XL, YL, ZL, FF)
        /* from the computed distances etc., compute the
           intermolecular forces and update the force (or
           acceleration) locations */

double XL[], YL[], ZL[], FF[];
{
    double G110[3], G23[3], G45[3], TT1[3], TT[3], TT2[3];
    double GG[15][3];

    /* compute local arrays G110, G23, G45, TT1, TT, TT2 and GG */
    ...

    /* lock locations for the molecule to be updated */
    lock(MolLock[mol % MAXMOLLOCKS]);
    VAR[mol].F[DEST][XDIR][O]  +=
            G110[XDIR] + GG[11][XDIR] +GG[12][XDIR]+C1*G23[XDIR];
    VAR[mol].F[DEST][XDIR][H1] +=
            GG[6][XDIR]+GG[7][XDIR]+GG[13][XDIR]+TT[XDIR]+GG[4][XDIR];
    VAR[mol].F[DEST][XDIR][H2] +=
            GG[8][XDIR]+GG[9][XDIR]+GG[14][XDIR]+TT[XDIR]+GG[5][XDIR];
    VAR[mol].F[DEST][YDIR][O]  +=
            G110[YDIR]+GG[11][YDIR]+GG[12][YDIR]+C1*G23[YDIR];
    VAR[mol].F[DEST][YDIR][H1] +=
            GG[6][YDIR]+GG[7][YDIR]+GG[13][YDIR]+TT[YDIR]+GG[4][YDIR];
    VAR[mol].F[DEST][YDIR][H2] +=
            GG[8][YDIR]+GG[9][YDIR]+GG[14][YDIR]+TT[YDIR]+GG[5][YDIR];
    VAR[mol].F[DEST][ZDIR][O]  +=
            G110[ZDIR]+GG[11][ZDIR]+GG[12][ZDIR]+C1*G23[ZDIR];
    VAR[mol].F[DEST][ZDIR][H1] +=
            GG[6][ZDIR]+GG[7][ZDIR]+GG[13][ZDIR]+TT[ZDIR]+GG[4][ZDIR];
    VAR[mol].F[DEST][ZDIR][H2] +=
            GG[8][ZDIR]+GG[9][ZDIR]+GG[14][ZDIR]+TT[ZDIR]+GG[5][ZDIR];
    unlock(MolLock[mol % MAXMOLLOCKS]);


    lock(MolLock[comp % MAXMOLLOCKS]);
    VAR[comp].F[DEST][XDIR][O] +=
          -G110[XDIR]-GG[13][XDIR]-GG[14][XDIR]-C1*G45[XDIR];
    VAR[comp].F[DEST][XDIR][H1] +=
          -GG[6][XDIR]-GG[8][XDIR]-GG[11][XDIR]-TT2[XDIR]-GG[2][XDIR];
    VAR[comp].F[DEST][XDIR][H2] +=
          -GG[7][XDIR]-GG[9][XDIR]-GG[12][XDIR]-TT2[XDIR]-GG[3][XDIR];
    VAR[comp].F[DEST][YDIR][O] +=
          -G110[YDIR]-GG[13][YDIR]-GG[14][YDIR]-C1*G45[YDIR];
    VAR[comp].F[DEST][YDIR][H1] +=
          -GG[6][YDIR]-GG[8][YDIR]-GG[11][YDIR]-TT2[YDIR]-GG[2][YDIR];
    VAR[comp].F[DEST][YDIR][H2] +=
          -GG[7][YDIR]-GG[9][YDIR]-GG[12][YDIR]-TT2[YDIR]-GG[3][YDIR];
    VAR[comp].F[DEST][ZDIR][O] +=
          -G110[ZDIR]-GG[13][ZDIR]-GG[14][ZDIR]-C1*G45[ZDIR];
    VAR[comp].F[DEST][ZDIR][H1] +=
          -GG[6][ZDIR]-GG[8][ZDIR]-GG[11][ZDIR]-TT2[ZDIR]-GG[2][ZDIR];
    VAR[comp].F[DEST][ZDIR][H2] +=
          -GG[7][ZDIR]-GG[9][ZDIR]-GG[12][ZDIR]-TT2[ZDIR]-GG[3][ZDIR];
    unlock(MolLock[comp % MAXMOLLOCKS]);
}           /* end of subroutine UPDATE_FORCES */
```

Figure 6.1: Computation of inter-molecular interactions in Water.

outside the mutex body. The resulting mutex body contains the minimal set of computations needed to maintain the semantics of the original code in Figure 6.1.

In a more recent version of the SPLASH suite, the Water application has been modified so that the code that computes inter-molecular interactions does not need this synchronization anymore (Woo et al. 1995). Therefore, when applied to the new version, the LICM optimization has no effect. The effect of reducing the size of mutual exclusion sections is only measurable if there exists a high lock overhead in the original program. In the case of Water, mutual exclusion sections are very small (the sections in Figure 6.1 are the two biggest ones) and total synchronization overhead can be reduced by solving larger problems (Singh et al. 1992).

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

The first experiment studies the performance effects of LICM as a function of synchronization overhead. As the number of time-steps increases, so does synchronization overhead. Table 6.1 shows the speedups obtained as a function of the number of time-steps and number of molecules simulated. Notice how the speedups obtained by LICM are lower when a larger number of molecules are simulated. This is caused by the larger computation to synchronization ratio in the larger problem. Also, by restricting the number of molecule locks available we are increasing lock contention. Naturally, as the number of available locks increases, the effects of LICM are diminished.

Since molecule locks are accessed more as the number of time-steps increases, the contention on these locks also increases. To measure lock contention we used the hardware timers provided by the system to measure the average delay of acquiring a lock. We then computed the average delay over the 10 molecule locks. This is shown in Table 6.2. This table shows how

```
UPDATE_FORCES(DEST, mol, comp, XL, YL, ZL, FF)
  double XL[], YL[], ZL[], FF[];
{
  ...

  t1 = *G110 + GG[11][0] + GG[12][0] + C1 * *G23;
  t2 = GG[6][0] + GG[7][0] + GG[13][0] + *TT + GG[4][0];
  t3 = GG[8][0] + GG[9][0] + GG[14][0] + *TT + GG[5][0];
  t4 = G110[1] + GG[11][1] + GG[12][1] + C1 * G23[1];
  t5 = GG[6][1] + GG[7][1] + GG[13][1] + TT[1] + GG[4][1];
  t6 = GG[8][1] + GG[9][1] + GG[14][1] + TT[1] + GG[5][1];
  t7 = G110[2] + GG[11][2] + GG[12][2] + C1 * G23[2];
  t8 = GG[6][2] + GG[7][2] + GG[13][2] + TT[2] + GG[4][2];
  t9 = GG[8][2] + GG[9][2] + GG[14][2] + TT[2] + GG[5][2];
  suif_tmp19 = &VAR[mol].F[7][0][1];
  suif_tmp21 = &VAR[mol].F[7][0][0];
  suif_tmp23 = &VAR[mol].F[7][0][2];
  suif_tmp25 = &VAR[mol].F[7][1][1];
  suif_tmp27 = &VAR[mol].F[7][1][0];
  suif_tmp29 = &VAR[mol].F[7][1][2];
  suif_tmp31 = &VAR[mol].F[7][2][1];
  suif_tmp33 = &VAR[mol].F[7][2][0];
  suif_tmp35 = &VAR[mol].F[7][2][2];

  lock(MolLock[mol % 216]);
  *suif_tmp19 = *suif_tmp19 + t1;
  *suif_tmp21 = *suif_tmp21 + t2;
  *suif_tmp23 = *suif_tmp23 + t3;
  *suif_tmp25 = *suif_tmp25 + t4;
  *suif_tmp27 = *suif_tmp27 + t5;
  *suif_tmp29 = *suif_tmp29 + t6;
  *suif_tmp31 = *suif_tmp31 + t7;
  *suif_tmp33 = *suif_tmp33 + t8;
  *suif_tmp35 = *suif_tmp35 + t9;
  unlock(MolLock[mol % 216]);

  ...
  /* Second mutex body removed for space considerations. */
}
```

Figure 6.2: Effect of LICM on the first mutex body of Figure 6.1.

| | 64 molecules (10 molecule locks) | | | 216 molecules (10 molecule locks) | | |
|---|---|---|---|---|---|---|
| Time steps | Unopt time (secs) | Opt time (secs) | Relative Speedup | Unopt time (secs) | Opt time (secs) | Relative Speedup |
| 70 | 157 | 144 | 1.09 | 1527 | 1463 | 1.04 |
| 80 | 183 | 171 | 1.07 | 1772 | 1763 | 1.00 |
| 100 | 235 | 219 | 1.07 | 2344 | 2285 | 1.02 |
| 120 | 296 | 269 | 1.10 | 2827 | 2809 | 1.00 |

Table 6.1: Speedups obtained by LICM on Water as a function of the number of simulation time-steps.

| | 64 molecules | | | 216 molecules | | |
|---|---|---|---|---|---|---|
| Time steps | Unoptimized avg delay ($\mu$secs) | Optimized avg delay ($\mu$secs) | Ratio | Unoptimized avg delay ($\mu$secs) | Optimized avg delay ($\mu$secs) | Ratio |
| 70 | 699 | 72 | 9.71 | 561 | 68 | 8.25 |
| 80 | 712 | 73 | 9.75 | 575 | 72 | 7.99 |
| 100 | 718 | 71 | 10.11 | 557 | 70 | 7.96 |
| 120 | 729 | 85 | 8.58 | 564 | 62 | 9.10 |

Table 6.2: Effects of LICM on lock contention in Water.

average lock contention on the molecule locks increases as a function of the number of simulation time-steps. Notice that although LICM reduces lock contention significantly, its impact on the runtime of the program may not be too noticeable if the ratio of computation to synchronization is high enough. Again notice how lock contention decreases with the larger problem size. This explains the diminished effects of LICM on large problems.

This implementation of Water contains another optimization that has been applied manually by the programmer: the simulation computes global sums that are first computed locally and then propagated to the global counter. To test the effects of MBL and LICM, we simplified these routines to perform all the computations on the shared variables directly. The intent of this experiment is to show that it is possible to automate common optimization patterns that experienced programmers implement manually.

Figure 6.3 shows a fragment of a routine that computes a reduction on the global variable `VIR`. After recognizing the reduction, the compiler applied MBL and LICM to obtain the equivalent and more efficient code in Figure 6.4.[2] This is virtually the same code included in the original Water application.

### Ocean

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

[2] We needed to annotate references to array `VAR` as non-conflicting to circumvent limitations in the compiler.

```
INTRAF()
{
  ...

  /* calculate summation of the product of the displacement and computed
     force for every molecule, direction, and atom */

  lock(gl->IntrafVirLock)

  for (mol = StartMol[ProcID]; mol < StartMol[ProcID+1];  mol++)
     for ( dir = XDIR; dir <= ZDIR; dir++)
        for (atom = 0; atom < NATOM; atom++)
           VIR += VAR[mol].F[DISP][dir][atom] * VAR[mol].F[FORCES][dir][atom];

  unlock(gl->IntrafVirLock)
} /* end of subroutine INTRAF */
```

Figure 6.3: Simplified version of function `INTRAF` in Water.

```
INTRAF()
{
  ...

  _local_VIR = 0.0;
  for (mol = StartMol[ProcID]; mol < StartMol[ProcID+1];  mol++)
     for (dir = 0; dir <= 2; dir++)
        for (atom = 0; atom < 3; atom++)
           _local_VIR = _local_VIR + VAR[mol].F[0][dir][atom] * VAR[mol].F[7][dir][atom];

  lock(gl->IntrafVirLock)
  VIR =  VIR + _local_VIR;
  unlock(gl->IntrafVirLock)
}
```

Figure 6.4: Effects of MBL and LICM on the code in Figure 6.3.

| Ocean size | Unoptimized time (sec) | Optimized time (sec) | Relative Speedup |
|---|---|---|---|
| $66 \times 66$ | 21 | 19 | 1.11 |
| $130 \times 130$ | 69 | 56 | 1.23 |
| $258 \times 258$ | 258 | 198 | 1.30 |
| $514 \times 514$ | 865 | 787 | 1.10 |

Table 6.3: Effects of MBL and LICM on Simple Ocean.

aggregated to the global sum.

To study the effect of MBL and LICM on this application, we re-wrote some routines in Ocean to use the simpler method of updating global sums. We named this new version Simple Ocean. The intention is to demonstrate how some of the optimizations that are traditionally performed manually by the programmer can be automated using the techniques developed in this thesis. Table 6.3 shows the performance improvements obtained by applying MBL and LICM to Simple Ocean. The program was executed on 8 processors with four different ocean sizes and a time-step of 180 seconds.

Procedure `slave` in Figure 6.5 contains a mutex body that updates a global sum (variable `psibi`). This version is different from the original in that the reduction is computed directly on the shared variable `psibi`. After reduction recognition and the application of MBL and LICM to the code in Figure 6.5, the compiler generated the equivalent and more efficient version of Figure 6.6. The resulting code is the same code for procedure `slave` included in the original Ocean application, but in this case the compiler performed the optimization, not the programmer.

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

```
void
slave ()
{
  ...
  /* update the shared variable psibi by summing all the psibis
     of the individual processes into it.  This is a simpler but
     more inefficient version of the original Ocean application. */
  lock (psibilock);

  if (procid == MASTER) {
    psibi = psibi + 0.25 * (wrk1->psib[0][0]);
  }
  if (procid == xprocs − 1) {
    psibi = psibi + 0.25 * (wrk1->psib[0][jm − 1]);
  }
  if (procid == nprocs − xprocs) {
    psibi = psibi + 0.25 * (wrk1->psib[im − 1][0]);
  }
  if (procid == nprocs − 1) {
    psibi = psibi + 0.25 * (wrk1->psib[im − 1][jm − 1]);
  }
  if (firstrow == 1) {
    for (j = firstcol; j <= lastcol; j++) {
      psibi = psibi + 0.5 * wrk1->psib[0][j];
    }
  }
  if ((firstrow + numrows) == im − 1) {
    for (j = firstcol; j <= lastcol; j++) {
      psibi = psibi + 0.5 * wrk1->psib[im − 1][j];
    }
  }
  if (firstcol == 1) {
    for (j = firstrow; j <= lastrow; j++) {
      psibi = psibi + 0.5 * wrk1->psib[j][0];
    }
  }
  if ((firstcol + numcols) == jm − 1) {
    for (j = firstrow; j <= lastrow; j++) {
      psibi = psibi + 0.5 * wrk1->psib[j][jm − 1];
    }
  }
  for (iindex = firstcol; iindex <= lastcol; iindex++) {
    for (i = firstrow; i <= lastrow; i++) {
      psibi = psibi + wrk1->psib[i][iindex];
    }
  }

  unlock (>psibilock);
  ...
}
```

Figure 6.5: Procedure `slave` in Simple Ocean.

```
void
slave ()
{
  ...
  _local_psibi = 0.0;



  if (procid == MASTER) {
    _local_psibi = _local_psibi + 0.25 * (wrk1->psib[0][0]);
  }
  if (procid == xprocs - 1) {
    _local_psibi = _local_psibi + 0.25 * (wrk1->psib[0][jm - 1]);
  }
  if (procid == nprocs - xprocs) {
    _local_psibi = _local_psibi + 0.25 * (wrk1->psib[im - 1][0]);
  }
  if (procid == nprocs - 1) {
    _local_psibi = _local_psibi + 0.25 * (wrk1->psib[im - 1][jm - 1]);
  }
  if (firstrow == 1) {
    for (j = firstcol; j <= lastcol; j++) {
      _local_psibi = _local_psibi + 0.5 * wrk1->psib[0][j];
    }
  }
  if ((firstrow + numrows) == im - 1) {
    for (j = firstcol; j <= lastcol; j++) {
      _local_psibi = _local_psibi + 0.5 * wrk1->psib[im - 1][j];
    }
  }
  if (firstcol == 1) {
    for (j = firstrow; j <= lastrow; j++) {
      _local_psibi = _local_psibi + 0.5 * wrk1->psib[j][0];
    }
  }
  if ((firstcol + numcols) == jm - 1) {
    for (j = firstrow; j <= lastrow; j++) {
      _local_psibi = _local_psibi + 0.5 * wrk1->psib[j][jm - 1];
    }
  }
  for (iindex = firstcol; iindex <= lastcol; iindex++) {
    for (i = firstrow; i <= lastrow; i++) {
      _local_psibi = _local_psibi + wrk1->psib[i][iindex];
    }
  }

  lock (psibilock);
  psibi = psibi + _local_psibi;
  unlock (psibilock);
  ...
}
```

Figure 6.6: Effects of MBL and LICM on the code in Figure 6.5.

### 6.2.2 Java Applications

We selected programs originally written in Java because we anticipated optimization opportunities due to the thread-safe nature of its libraries. Although the concurrency and synchronization model used in Java are different from the assumptions made in this thesis, we think that it might be possible to apply these ideas to the Java environment. We study the potential benefits of LICM and Lock Picking in the context of concurrent and sequential Java programs. To illustrate the effects of LICM we show two parallel applications: parallel sorting and parallel matrix multiply.

**PSRS (Parallel Sorting by Regular Sampling)** is an explicitly parallel sorting algorithm (Shi and Schaeffer 1992) that samples the data to generate pivot elements that evenly distribute data among the processors. Each process uses a sequential sort algorithm to sort its own partition. The resulting data is then merged to obtain the final sorted list. The original Java program was implemented using the JGL (Java Generic Library) class library which provides a sequential quicksort algorithm and classes for creating abstract arrays. Since JGL is a thread-safe library, many of its classes and methods are synchronized. In this particular application, some of the synchronization is unnecessary. When a process is sorting, it never reads or writes outside its designated partition. Therefore, references to the shared array are lock independent and can be hoisted using LICM.

**Matrix multiply (MM)**: input matrix $A$ is blocked into non-overlapping sections which are assigned to a different process. Each process writes to a different cell of the result matrix $C$ and makes read-only references to the input matrices $A$ and $B$. No synchronization is required in this algorithm but the class libraries make use of synchronized methods to read and write to the different arrays.

#### Java Implementation

We performed two sets of experiments with these applications. First, we modified the Java implementation of these algorithms to emulate the effect of

| List size | Unoptimized time (secs) | Optimized time (secs) | Relative Speedup |
|---:|---:|---:|---:|
| 50,000 | 13 | 11 | 1.18 |
| 100,000 | 24 | 13 | 1.85 |
| 500,000 | 123 | 51 | 2.41 |
| 750,000 | 187 | 75 | 2.50 |
| 1,000,000 | 276 | 113 | 2.44 |
| 1,250,000 | 336 | 141 | 2.38 |

Table 6.4: Effects of LICM on the original Java implementation of the PSRS sorting algorithm (8 processors).

| Matrix size | Unoptimized time (secs) | Optimized time (secs) | Relative Speedup |
|---:|---:|---:|---:|
| 64×64 | 4 | 4 | 1.00 |
| 128×128 | 9 | 8 | 1.13 |
| 256×256 | 33 | 17 | 1.94 |
| 512×512 | 172 | 100 | 1.72 |
| 1024×1024 | 1484 | 810 | 1.83 |

Table 6.5: Effects of LICM on the Java implementation of matrix multiplication (8 processors).

Lock-Independent Code Motion. Essentially we transformed two *synchronized* methods into regular methods. In the case of PSRS, this is the `at` method in the JGL `ObjectArray` class. In the case of matrix multiply, this is the `intAt` method in the JGL `IntArray` class. Both methods are automatically synchronized by the library but in these applications, such synchronization is unnecessary because the different threads never make conflicting references to common array locations. Tables 6.4 and 6.5 show the performance improvements obtained by applying LICM to the PSRS and matrix multiply applications respectively. The programs were executed on a dedicated 8-processor SGI PowerChallenge.

Notice that this seemingly simple transformation has a noticeable impact on performance. On average, the optimized version of PSRS performs twice as fast as the unoptimized version. This is a strong indication of the potential that these types of techniques have on high-level languages like Java. We

| List size | Unoptimized time (secs) | Optimized time (secs) | Relative Speedup |
|---|---|---|---|
| 50,000 | 197 | 67 | 2.94 |
| 100,000 | 27 | 10 | 2.70 |
| 500,000 | 170 | 62 | 2.74 |
| 750,000 | 299 | 76 | 3.93 |
| 1,000,000 | 407 | 160 | 2.54 |
| 1,250,000 | 618 | 359 | 1.72 |

Table 6.6: Effects of LICM on the C implementation implementation of the PSRS sorting algorithm (2 processors).

| Matrix size | Unoptimized time (secs) | Optimized time (secs) | Relative Speedup |
|---|---|---|---|
| 64×64 | 2 | 1 | 2.00 |
| 128×128 | 12 | 5 | 2.40 |
| 256×256 | 82 | 22 | 3.73 |
| 512×512 | 638 | 163 | 3.91 |
| 1024×1024 | 5077 | 1276 | 3.98 |

Table 6.7: Effects of LICM on the C implementation of matrix multiplication (2 processors).

obtained similar improvement factors in matrix multiply. For small matrices, both versions performed roughly the same but as the size of the matrices grows, the effects of LICM tend to be more significant.

### C Implementation

In the second experiment we converted the Java programs into C using the Toba translator (Proebsting et al. 1998). Since the compiler cannot handle the code generated by Toba automatically, we manually applied the optimizations to the generated C programs.

These experiments were executed on a different machine because the Toba runtime libraries did not work on the PowerChallenge. We used a dedicated two-processor SGI Octane for the C implementation of PSRS and matrix multiply. Tables 6.6 and 6.7 show the results obtained for PSRS and matrix multiply respectively.[3]

Although the execution environment for both experiments is different, we observed an interesting fact. The performance improvements obtained in the C version of these programs are better than those obtained in their Java counterparts. In the case of matrix multiply, these improvements are significantly better. Using the SpeedShop profiling tool available on SGI machines we determined that in some cases the unoptimized programs spent up to 30% of their time trying to enter the monitor protecting the synchronized methods. In these experiments we only used two threads to execute the application and the profiling tool did not report any other thread activity. There are two explanations for this excessive synchronization overhead: (a) the implementation of locks in Toba is inferior to that of Java, or, (b) the individual threads in the C version are so much faster than the Java version that once they leave the critical section they quickly try to acquire the lock again.

The profiling logs show that the function acting as the entry point to the monitor spends roughly 70% of its time spinning on the lock variable that implements the monitor. We conclude that the excessive synchronization overhead of the C version is mostly due to lock contention. However, as the results in the next section show, the lock implementation is also important as it may also affect the performance of sequential programs.

### Sequential Java Programs

In this section we show how our transformation techniques might benefit sequential programs. Since the CSSAME form for a sequential program has no $\pi$ functions, the Lock-Picking transformation can easily traverse all the mutex bodies in the program removing the synchronization operations. To illustrate the potential benefits of this optimization we used a set of benchmark programs that exercise different components of the JGL abstract class library. There are three programs:

(1) *Array* exercises common operations on abstract arrays: get, put,

[3] We also ran the Java version on the SGI Octane. The speedup ratios were the same as those shown in Tables 6.4 and 6.5.

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

Table 6.8: Effect of Lock-Picking (LP) on sequential Java programs.

iterate, clear and remove.

(2) *Map* exercises common operations on hash tables: add, find, remove and clear.

(3) *Sort* compares the sorting algorithm provided by JGL against a hand-coded quicksort algorithm.

Table 6.8 shows the improvements obtained by applying lock-picking to these programs. We executed both the Java and C versions of these programs; in both cases the results were similar. In general, we obtained performance improvements between 10% and 20% when lock-picking was applied.

The performance gains obtained by removing the unnecessary locks are directly related to this particular implementation of mutual exclusion. Since these are sequential programs, all the synchronization overhead is caused by the actual call to `lock` and `unlock`. There is no lock contention. An alternative to removing the locks would have been to use a more efficient mutual exclusion synchronization implementation (Bacon et al. 1998). We are convinced that a combination of compiler optimizations and efficient lock implementations is the best approach in these cases.

### 6.2.3 Other Applications

We also studied two applications included in the TreadMarks DSM system (Keleher et al. 1994), namely the Traveling Salesman Problem (TSP) and a parallel quicksort implementation (QS). Lock contention is not a problem in these two implementations. The LICM transformation made some minor modifications to the mutex structures in these programs that did not affect the runtime performance of either program. However, the analysis techniques helped us locate data races and locking irregularities.

This TSP implementation takes advantage of the weak memory semantics in TreadMarks. Since updates to shared variables are only visible at synchronization points, TSP makes unprotected references to shared variables without causing data races. However, with the strong memory semantics used in our model it was necessary to privatize some global variables to avoid data races in the program. While none of the synchronization transformations found opportunities for optimization, the analysis of mutex sections detected an irregularity in the original program: one of the procedures was tripping over a lock. (i.e., the same lock was being acquired more than once). The compiler also found several data races triggered by conflicting data references outside mutex bodies.

The quicksort implementation used another implementation “trick” to force propagating the update to a flag variable shared between the worker threads. The code fragment in Figure 6.7 shows how this is implemented. Note that this is the same code from Figure 3.5. We have reproduced it here for easier reference. To propagate an update of the shared variable `pause_flag` in TreadMarks, it is necessary to use `lock` and `unlock` operations to force a consistency operation in the DSM system. However, using the stronger memory semantics assumed in our model the compiler determined that since the mutex body for lock variable `pause_lock` was always nested inside a mutex body for lock variable `TSL`, it could be eliminated. Therefore, the lock operations at lines 13, 15, 21 and 23 were all removed by the compiler.

## 6.3 Conclusions

The programs described in this chapter represent two different types of explicitly parallel programs which we call high-level and low-level parallelism. The first group (low-level parallelism) are programs developed in environments where the user has complete control over the parallel and synchronization structure of the program. Typically, these programs have been manually

```
#define NPROCS 5
#define DONE -1

int PopWork(TaskElement *task)
{
    lock(TSL);

    while (TaskStackTop == 0) {
        if (++NumWaiting == NPROCS) {
            /* All the threads are waiting for work.
             * We are done.
             */
            lock(pause_lock);
            pause_flag = 1;
            unlock(pause_lock);

            unlock(TSL);
            return DONE;
        } else {
            if (NumWaiting == 1) {
                lock(pause_lock);
                pause_flag = 0;
                unlock(pause_lock);
            }

            unlock(TSL)

            /* Wait for work. This is the only
             * statement not protected by TSL.
             */
            while (!pause_flag) ; /* busy-wait */

            lock(TSL);

            if (NumWaiting == NPROCS) {
                unlock(TSL);
                return DONE;
            }
            --NumWaiting;
        }
    } /* while task-stack empty */

    /* Pop a piece of work from the stack */
    TaskStackTop--;
    task->left  = TaskStack[TaskStackTop].left;
    task->right = TaskStack[TaskStackTop].right;

    unlock(TSL);

    return 0;
}
```

Figure 6.7: Nested mutex bodies in function *PopWork()*.

optimized by experienced programmers who make an effort to minimize mutual exclusion sections as much as possible.

The second group (high-level parallelism) includes systems that offer thread-safe libraries and programs developed in programming environments that generate generic code templates on behalf of the user. These applications can contain conservative mutual exclusion structures that may hurt performance unnecessarily.

We have shown that the techniques developed in this thesis can have a significant impact on the performance of high-level parallel applications. Furthermore, we have also shown that performance gains can be obtained in low-level parallel programs. We have demonstrated that it is possible to automate some of the manual transformations that programmers routinely make to minimize mutual exclusion sections.

We consider these techniques a first step to fully exploiting the optimization possibilities in explicitly parallel programs. Currently, our technology allows the compiler to perform some of the same optimizations that an experienced programmer can do manually. In the future we expect this situation to be reversed: compilers for parallel programs will make more and better transformations that cannot be easily duplicated by programmers.

# Chapter 7

# Conclusions and Future Work

## 7.1 Summary of Contributions

Explicitly parallel programs for shared memory architectures offer new challenges to an optimizing compiler; multiple threads of activity in a parallel program can alter data and control dependencies in ways that existing compiler technology cannot detect. The new analysis and optimization techniques developed in this thesis represent a significant step towards improving the capabilities of compilers for explicitly parallel programs. We expect these techniques to be particularly useful in the context of high-level concurrent or thread-based languages. Of particular importance in these environments is the ability of the compiler to understand synchronization operations which can be a source of substantial overhead in some applications.

Although compilers for parallel computing have been the focus of intense research and development, most efforts have been concentrated on the automatic transformation of sequential programs into their parallel counterpart. Parallelizing and vectorizing compilers take a sequential program and turn it into their equivalent parallel version. The topic of analyzing explicitly parallel code for the purpose of optimization has received scant attention. The CSSAME framework proposed in this thesis provides the necessary tools for a compiler to reason about and optimize an explicitly parallel program containing synchronization.

### 7.1.1 Analysis

The CSSAME form provides a comprehensive data-flow framework for analyzing explicitly parallel programs. Inter-process interactions via data sharing and synchronization constructs are taken into consideration. In this thesis we have shown how to build the fundamental data structures and we have used them to find basic information like reaching definitions, reached uses and mutual exclusion synchronization patterns. We have also shown how existing synchronization analyses can be incorporated into the base framework to augment the non-concurrency information needed to disregard shared memory interactions that are made impossible by synchronization restrictions.

The memory semantics considered by this work represent the most general scenario from the point of view of an optimizing compiler, since every update to a shared memory variable is immediately visible to other threads, the compiler can make no assumptions about the value of the variable at any point in the program.

Weaker memory models allow shared memory updates to be propagated at later time. This is typically used in Distributed Shared Memory systems to optimize traffic through the memory interconnect. Shared memory is updated after certain events like synchronization points or via specific memory barrier instructions inserted in the program. Incorporating these semantics into the CSSAME construction algorithm may lead to fewer $\pi$ functions which in turn will allow more aggressive transformations.

Synchronization is an important component of every parallel program. An optimizing compiler must be aware of synchronization constructs in a parallel program for two fundamental reasons:

1. **Validation**. We have shown how the compiler can warn the user about illegal or inconsistent synchronization patterns when using mutual exclusion. This can be augmented with other existing synchronization analysis methods that can detect deadlocks and race conditions in a program. Although it has been shown that some of these methods are exponentially expensive, simplified versions can still be used to provide compile-time warnings to the user.

2. **Optimization**. Synchronization can provide several optimization opportunities. The main effect of synchronization is the elimination of some shared memory interactions that may be preventing a transformation. It is also possible to detect overly restrictive synchronization patterns like nested mutex structures that can be eliminated (Section 5.3).

### 7.1.2 Optimization

We have shown how the CSSAME form is unique in allowing new optimization opportunities by taking advantage of the semantics imposed by synchronization. Two types of optimization are possible: the adaptation of existing sequential techniques and the direct optimization of parallel and synchronization structures in the program.

#### Adapting Sequential Techniques

The reduction of memory conflicts across threads can improve the effectiveness of adapted scalar optimization strategies like constant propagation. We have adapted a sequential dead-code elimination algorithm. In general, the process of adapting an existing sequential technique is mainly an implementation issue, especially if the technique is SSA based.

The concurrent version needs to consider $\pi$ functions in addition to $\phi$ functions. Also, cost models might need to be altered. For instance, in common sub-expression elimination, if a subexpression is common across several threads it might be cheaper to make each thread compute the expression instead of pushing it up into a sequential section of the program.

#### Optimizing the Structure of a Parallel Program

In this thesis we have introduced three new optimization techniques that are specifically targeted at explicitly parallel programs: *lock picking* examines and removes unnecessary `lock` and `unlock` operations, *lock-independent code motion* moves code that does not need to be locked outside critical sections and *mutex body localization* converts shared memory references into local memory references. Although we do not expect experienced programmers to write overly restrictive synchronization patterns, high-level systems like Java make use of generic thread-safe libraries that must make conservative assumptions about the application's context. Therefore, when considered within the context of a particular program it might turn out that many synchronization operations are not necessary. We have shown how techniques like lock picking and lock independent code motion benefit these applications.

We consider these techniques a significant step towards facilitating the adoption of high-level systems with language-supported parallelism and synchronization. These systems typically provide powerful abstractions that make parallel programming easier, but those same abstractions often hinder performance. Experienced programmers recognize these limitations and manually circumvent them by removing abstraction layers to speed-up their code. This defeats the purpose of having the high-level abstractions and it is something that should be addressed by the compiler, not the user.

## 7.2 Future Work

Our long-term goal is to achieve the same level of sophistication in compilers for explicitly parallel languages as that of current compiler technology for sequential languages. The development of a complete compilation/performance tuning system for explicitly parallel programs is a massive multi-year project. In this thesis we have presented the base framework for such a project. The following sections discuss future work directions and our vision for what an optimizing compiler for parallel languages should provide.

### 7.2.1 Parallelism

There are many ways of specifying parallel activity in a program. The primitives used in this work, `cobegin/coend` and `parloop`, were selected because of their conceptual simplicity and expressive power. They can be used to describe a wide variety of task and data parallel programs.

```
main()
{
  /* Call function f() to execute
     concurrently with the main
     thread.
   */
  fork(f);

  do_work();

  /* Wait for child thread. */
  wait();
}

f()
{
  do_work();
}
```

Figure 7.1: Expressing parallel activity using `fork`.

Other mechanisms can be incorporated into the framework. For instance, many platforms provide a `fork` system call that takes a function name as its argument. When invoked, `fork` launches a new thread to execute the given function in parallel. The calling thread continues to execute concurrently with the newly launched thread (Figure 7.1).

The important information to be gathered is the concurrency relation given by Algorithm 3.2. Given two flowgraph nodes $a$ and $b$, the concurrency analysis determines whether $a$ and $b$ may execute concurrently. This accuracy of the concurrency information is subject to the assumptions made by the analysis method, but it must be conservatively correct. When it is not clear whether two nodes may execute concurrently or not, the analysis must assume that they will.

In some cases, gathering this information may be a simple task. For instance, in a high-level programming environment like Enterprise (Schaeffer et al. 1993), all the concurrency information is contained in an external graph representation of the program modules which can be readily used by the compiler. In other cases, this might be more difficult. In the case of the example program in Figure 7.1 the analysis should traverse the flow graph for each function marking for each statement which other statements can execute concurrently. Initial support for the *pthreads* library (Lewis and Berg 1998) has been implemented in our compiler.

### 7.2.2 Synchronization

Synchronization analysis is a fundamental component of every optimizing compiler for explicitly parallel languages. Information gathered from the synchronization patterns in the program can be used to warn the user about potential problems and to make optimization decisions.

It is important to observe that some synchronization mechanisms offer little non-concurrency information to a static analyzer. Consider for instance *counting semaphores* (Tanenbaum 1992). Counting semaphores are used to allow a limited number of threads to have concurrent access to the same resource pool. These semantics do not facilitate the elimination of $\pi$ functions as is the case with `lock`, `barrier` and `set/wait` constructs. However, if the compiler can determine that a particular counting semaphore is always initialized to 1 then it can be treated like a mutual exclusion operation.

Synchronization can also be achieved without using special constructs. A typical example is given in Figure 7.2. Thread $T_1$ will not start executing until thread $T_0$ sets variable *busy* to 0. Although detecting this pattern might be more involved than recognizing synchronization primitives, it still could be incorporated and its effects would be the same as any other mutual exclusion construct. Both calls to function *compute*() in this example will be non-concurrent.

### 7.2.3 Other Memory Models

Different memory models have an impact on the placement of $\pi$ functions because they allow different memory interleavings than the semantics considered in this thesis. Earlier SSA frameworks for explicitly parallel programs were based on `copy-in/copy-out` semantics, a weaker form of consistency that guarantees updates at certain synchronization points (Srinivasan et al. 1993).

We plan to adapt the CSSAME infrastructure to different memory models.

```
main()
{
  busy = 1;
  cobegin {
    T0: begin
      compute();
      busy = 0;
    end

    T1: begin
      /* busy-wait until T0 has computed */
      while ( busy == 1 )
        ; /* busy wait */

      compute();
    end
  }
}
```

Figure 7.2: Mutual exclusion synchronization without locks.

Currently we are investigating release-consistent models (Keleher et al. 1994). In a release-consistent memory, updates to shared variables are only visible at synchronization points. This may lead to the elimination of more $\pi$ functions which in turn allow more aggressive optimizations.

### 7.2.4 Dependency Analysis

Results obtained in vectorizing and parallelizing compilers are also important in a compiler for explicitly parallel programs. In particular, the dependency analysis techniques developed for vectorizing and parallelizing compilers are an invaluable tool to fine-tune information about shared array references. Recent work proposes adapting a sequential array SSA form to the parallel case (Collard 1999).

### 7.2.5 Other Optimizations

#### Partial Redundancy Elimination (PRE)

Chow *et al.* developed an SSA-based partial redundancy elimination algorithm for sequential programs called SSAPRE (Chow et al. 1997).

```
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 thread propagation.

```
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 thread propagation.

Figure 7.3: Thread propagation optimization.

The transformation builds SSA information for selected sub-expressions. Expressions are assigned to hypothetical temporaries and the SSA information is built on those temporaries. Whenever one of the operands of the expression is modified, the associated temporary is also considered modified. Adapting SSAPRE to the parallel case involves building CSSAME information for the temporaries and treating them like any other variable in the program.

### Thread Propagation

Thread Propagation is a 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 7.3(a). The first three lines of the 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 7.3(b) shows the results of applying the thread propagation 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 $T_1$.

### Lock Partitioning

Lock partitioning 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 should 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.

## 7.3 Conclusions

An optimizing compiler for explicitly parallel languages must be able to handle different types of parallelism, synchronization constructs, and shared memory semantics. For instance, the compiler should recognize different synchronization constructs and adjust the data-flow representation appropriately. In this thesis we developed an SSA-based framework for analyzing these three elements. Regardless of the chosen analysis framework, it is important that it incorporates these three elements. Otherwise, decisions based on this analysis might yield erroneous transformations.

Optimizing transformations can be categorized as either adaptations of traditional sequential optimizations from or techniques that target one of the three elements mentioned above: parallelism, synchronization and shared memory semantics. In this thesis we have concentrated on the optimization of mutual exclusion synchronization. Using the prototype compiler that we are building, we will continue to investigate new analysis and optimization techniques for explicitly parallel programs.

# Bibliography


Aho, A. V., R. Sethi, and J. Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Second. Reading, MA: Reading, Mass.: Addison-Wesley.

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

Blume, W. and R. Eigenmann. 1992. "Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs." *IEEE Transactions on Parallel and Distributed Systems* 3 (6): 643–656 (November).

Brandis, M. M. and H. Moessenboeck. 1994. "Single-Pass Generation of Static Single-Assignment Form for Structured Languages." *ACM Transactions on Programming Languages and Systems* 16 (6): 1684–1698 (November).

Callahan, D., K. Kennedy, and J. Subhlok. 1990, March. "Analysis of Event Synchronization in a Parallel Programming Tool." *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* Seattle, WA, 21–30.

Chow, F., S. Chan, R. Kennedy, S.-M. Liu, R. Lo, and P. Tu. 1997. "A New Algorithm for Partial Redundancy Elimination based on SSA Form." *ACM SIGPLAN '97 Conference on Programming Language Design and Implementation.* Las Vegas.

Collard, J.F. 1999, September. "Array SSA for Explicitly Parallel Programs." *Proceedings of Euro-Par '99.*

Cytron, R., J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. 1991. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph." *ACM Transactions on Programming Languages and Systems* 13 (4): 451–490 (October).

Diniz, P. and M. Rinard. 1998. "Lock Coarsening: Eliminating Lock Overhead in Automatically Parallelized Object-based Programs." *Journal of Parallel and Distributed Computing* 49 (2): 218–244 (March).

Eigenmann, R. and W. Blume. 1991, August. "An Effectiveness Study of Parallelizing Compiler Techniques." *1991 International Conference on Parallel Processing.* St. Charles, IL.

Emrath, P. A., S. Ghosh, and D. A. Padua. 1992. "Detecting Nondeterminacy in Parallel Programs." *IEEE Software* 9 (1): 69–77 (January).

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

Gupta, M. and P. Banerjee. 1992. "Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers." *IEEE Transactions on Parallel and Distributed Systems* 3 (2): 179–193 (March).

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

Helmbold, D. P. and C. E. McDowell. 1994, September. "A taxonomy of race detection algorithms." Technical Report UCSC-CRL-94-35, University of California, Santa Cruz.

Hendren, L. 2000, February. "Personal communication."

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

Jeremiassen, T. and S. Eggers. 1994, August. "Static Analysis of Barrier Synchronization in Explicitly Parallel Systems." *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT).* Montreal, Canada.

Johnson, R., D. Pearson, and K. Pingali. 1994, June. "The Program Structure Tree: Computing Control Regions in Linear Time." *ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.* Orlando, Florida, 171–185.

Keleher, P., A. Cox, S. Dwarkadas, and W. Zwaenepoel. 1994, January. "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems." *Proceedings of the 1994 Winter USENIX Conference.*

Knoop, J., B. Steffen, and J. Vollmer. 1996. "Parallelism for Free: Efficient and Optimal Bitvector Analyses for Parallel Programs." *ACM Transactions on Programming Languages and Systems* 18 (3): 268–299 (May).

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

Lee, J., S. Midkiff, and D. A. Padua. 1997a, July. “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.

———. 1997b, August. “Concurrent Static Single Assignment Form and Constant Propagation for Explicitly Parallel Programs.” *Proceedings of the Tenth Workshop on Languages and Compilers for Parallel Computing.*

———. 1998. “A Constant Propagation Algorithm for Explicitly Parallel Programs.” *International Journal of Parallel Programming* 26 (5): 563–589.

Lee, J., D. A. Padua, and S. Midkiff. 1999, May. “Basic Compiler Algorithms for Parallel Programs.” *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* Atlanta, GA.

Lewis, Bil and Daniel J. Berg. 1998. *Multithreaded programming with pthreads.* 2550 Garcia Avenue, Mountain View, CA 94043, USA: Sun Microsystems.

Masticola, S. and B. Ryder. 1993, May. “Non-concurrency Analysis.” *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* San Diego, CA, 129–138.

Mellor-Crummey, J. M. and M. L. Scott. 1991. “Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors.” *ACM Transactions on Computer Systems* 9 (1): 21–65 (February). Earlier version published as TR 342, URCSD, April 1990, and COMP TR90-114, Center for Research on Parallel Computation, Rice UNIV, May 1990.

Midkiff, S. P. and D. A. Padua. 1990, August. “Issues in the Optimization of Parallel Programs.” *1990 International Conference on Parallel Processing*, Volume II. St. Charles, Ill., 105–113.

Muchnick, S. 1997. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers.

Netzer, R. H. B and B. P. Miller. 1990, August. “On the complexity of event ordering for shared memory parallel program executions.” *1990 International Conference on Parallel Processing*, Volume II Software. 93–104.

North, S. C. and E. Koutsofios. 1994, May. "Application of Graph Visualization." *Proceedings of Graphics Interface '94*. Canadian Information Processing Society Banff, Alberta, Canada, 235–245. URL: http://www.research.att.com/~north/graphviz/.

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

Proebsting, T. A., G. Townsend, P. Bridges, J. H. Hartman, T. Newsham, and S. A. Watterson. 1998. "Toba: Java For Applications — A Way Ahead of Time (WAT) Compiler." Technical Report, Department of Computing Science, The University of Arizona. URL: http://www.cs.arizona.edu/sumatra/toba/.

Pugh, W. and D. Wonnacott. 1992, June. "Eliminating False Data Dependences using the Omega Test." *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*. San Francisco, CA. URL: http://www.cs.umd.edu/projects/omega/.

Schaeffer, J., D. Szafron, G. Lobe, and I. Parsons. 1993. "The Enterprise Model for Developing Distributed Applications." *IEEE Parallel and Distributed Technology* 1 (3): 85–96.

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

Shi, H. and J. Schaeffer. 1992. "Parallel Sorting by Regular Sampling." *Journal of Parallel and Distributed Computing* 14 (4): 361–372.

Singh, J., W. Weber, and A. Gupta. 1992. "SPLASH: Stanford Parallel Applications for Shared-Memory." *Computer Architecture News* 20 (1): 5–44 (March).

Sreedhar, V. C. and G. R. Gao. 1995, January. "A Linear Time Algorithm for Placing $\phi$-nodes." *22nd Annual ACM Symposium on Principles of Programming Languages*. New York, NY, USA: ACM Press, 62–73.

Srinivasan, H., J. Hook, and M. Wolfe. 1993, January. "Static Single Assignment for Explicitly Parallel Programs." *20th Annual ACM Symposium on Principles of Programming Languages*. Charleston, S.C., 16–28.

Tanenbaum, A. S. 1992. *Modern Operating Systems*. Englewood Cliffs, NJ 07632: Prentice Hall.

Unrau, R., O. Krieger, B. Gamsa, and M. Stumm. 1994. "Experiences with Locking in a NUMA Multiprocessor Operating System Kernel."

*Proceedings for the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI '94)*. 139–152.

Wegman, M. and K. Zadeck. 1991. "Constant Propagation with Conditional Branches." *ACM Transactions on Programming Languages and Systems* 13 (2): 181–210 (April).

Whaley, J. and M. Rinard. 1999, November. "Compositional Pointer and Escape Analysis for Java Programs." *Proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*.

Wilson, R. et al. 1994. "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers." *ACM SIGPLAN Notices* 29 (12): 31–37 (December).

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

Woo, S. C., M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. 1995, June. "The SPLASH-2 Programs: Characterization and Methodological Considerations." *22nd International Symposium on Computer Architecture*. 24–36.