# Debugging Multithreaded Programs that Incorporate User-Level Locking

by

Andrew F. Stark

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1998

Author . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Department of Electrical Engineering and Computer Science
May 22, 1998

Certified by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Charles E. Leiserson
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

# Debugging Multithreaded Programs that Incorporate User-Level Locking

by

Andrew F. Stark

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 1998, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

## Abstract

A multithreaded program with a bug may behave nondeterministically, and this nondeterminism typically makes the bug hard to localize. This thesis presents a debugging tool, the Nondeterminator-2, which automatically finds certain nondeterminacy bugs in programs coded in the Cilk multithreaded language. Specifically, the Nondeterminator-2 finds "dag races," which occur when two logically parallel threads access the same memory location while holding no locks in common, and at least one of the accesses writes the location.

The Nondeterminator-2 contains two dynamic algorithms, All-Sets and Brelly, which check for dag races in the computation generated by the serial execution of a Cilk program on a given input. For a program that runs serially in time $T$, accesses $V$ shared memory locations, uses a total of $n$ locks, and holds at most $k \ll n$ locks simultaneously, All-Sets runs in $O(n^k T\,\alpha(V,V))$ time and $O(n^k V)$ space, where $\alpha$ is Tarjan's functional inverse of Ackermann's function. The faster Brelly algorithm runs in $O(kT\,\alpha(V,V))$ time using $O(kV)$ space and can be used to detect races in programs intended to obey the "umbrella" locking discipline, a programming methodology that precludes races.

In order to explain the guarantees provided by the Nondeterminator-2, we provide a framework for defining nondeterminism and define several "levels" of nondeterministic program behavior. Although precise detection of nondeterminism is in general computationally infeasible, we show that an "abelian" Cilk program, one whose critical sections commute, produces a determinate final state if it is deadlock free and if it can generate a dag-race free computation. Thus, the Nondeterminator-2's two algorithms can verify the determinacy of a deadlock-free abelian program running on a given input.

Finally, we describe our experiences using the Nondeterminator-2 on a real-world radiosity program, which is a graphics application for modeling light in diffuse environments. With the help of the Nondeterminator-2, we were able to speed up the entire radiosity application 5.97 times on 8 processors while changing less than 5 percent of the code. The Nondeterminator-2 allowed us to certify that the application had no race bugs with a high degree of confidence.

Thesis Supervisor: Charles E. Leiserson
Title: Professor of Computer Science and Engineering

# Acknowledgments

Portions of this thesis represent joint work with Guang-Ien Cheng, Mingdong Feng, Charles Leiserson, and Keith Randall, and I am indebted to all of them for their help. Among many other things, Ien provided remarkable insight into the workings of the SP-BAGS algorithm. Without complaint, Keith spent many a late night with me working on proofs as we encountered more and more difficulties in them. Mingdong inspired my original interest in the design and implementation of the Nondeterminator. Finally, I would like to thank Charles not only for his contribution to this research, but also for his never-ending patience, encouragement, and advice.

I would also like to thank the other members of the Cilk group for their contributions, and especially Matteo Frigo for many helpful suggestions. It has been a privilege to work with such a talented group of people.

The research in this thesis was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Grants N00014-94-1-0985 and F30602-97-1-0270. Also, I would like to thank Sun Microsystems for the use of the Xolas UltraSPARC SMP cluster.

It would be remiss of me not to thank Tom Pinckney and Scott Paxton, who convinced me (by example) to start and finish my thesis on time. Thanks to Chris McKinney for living in a different time zone, which gave me someone to talk to at 3:00 a.m. Also, thanks to Adam Harter for thoughtfully occupying my TV so it couldn't distract me.

And, of course, thanks to my parents, without whose support and encouragement I doubtlessly would be sleeping on the casino floor in Reno instead of debugging parallel programs.

# Contents

# List of Figures

# Chapter 1

# Introduction

When parallel programs have bugs, they can be nondeterministic, meaning that different executions produce different behaviors. In this thesis, we present a debugging tool, the Nondeterminator-2, which automatically finds nondeterminacy bugs in parallel programs. We give a theoretical model of nondeterminism that precisely explains the guarantees provided by the Nondeterminator-2. We further demonstrate the effectiveness of this debugging tool by showing how it was used to parallelize a complex, real-world application.

## Nondeterminism

Because of the vagaries of timings of multiple processors, parallel programs can be nondeterministic. Nondeterminism poses a serious challenge for debugging, because reproducing the situation that caused a particular bug can be difficult. Also, verifying that a program works correctly in one scheduling does not preclude the possibility of bugs in future executions.

In this thesis, we develop techniques for debugging parallel programs coded in the Cilk language. The Cilk [3, 4, 7, 16, 23] project is designed to make it easy for programmers to write efficient parallel programs. Parallel computing has long been an area of research, but it has yet to reach the "mainstream" world of professional programmers, even though parallel machines are becoming more available. Tradi-

```
int x;                      cilk void foo1()
                            {
cilk int main()               x += 2;
{                           }
  x = 2;
  spawn foo1();             cilk void foo2()
  spawn foo2();             {
  printf("%d", x);            x *= 3;
  return 0;                 }
}
```

**Figure 1-1**: A nondeterministic Cilk program. The spawn statement in a Cilk program creates a parallel subprocedure, and the sync statement provides control synchronization to ensure that all spawned subprocedures have completed.

| | | | Temporary values | |
|---|---|---|---|---|
| Time ↓ | Processor 1 | Processor 2 | $tmp_1$ | $tmp_2$ |
| | | | – | – |
| | $tmp_1 \leftarrow x$ | | 2 | – |
| | | $tmp_2 \leftarrow x$ | 2 | 2 |
| | $tmp_1 \leftarrow tmp_1 + 2$ | | 4 | 2 |
| | | $tmp_2 \leftarrow tmp_2 * 3$ | 4 | 6 |
| | $x \leftarrow tmp_1$ | | 4 | 6 |
| | | $x \leftarrow tmp_2$ | 4 | 6 |

**Figure 1-2**: An example of the machine instructions comprising updates to a shared variable x being interleaved. The final value of x in this particular execution is 6.

tional techniques for parallelization typically require programmers to have intimate knowledge of the workings of their parallel architectures. Cilk alleviates this problem by allowing programmers to code in the Cilk language, which is a simple extension to the programming language C [24]. The Cilk runtime system then automatically and efficiently runs this code on multiprocessor machines.

Cilk programs can still have nondeterminacy bugs, however. Figure 1-1 shows a Cilk program that behaves nondeterministically. The procedures foo1 and foo2 run in parallel, resulting in parallel access to the shared variable x. The value of x printed by main is 12 if foo1 happens to run before foo2, but it is 8 if foo2 happens to run before foo1. Additionally, main might also print 4 or 6 for x, because the statements in foo1 and foo2 are composed of multiple machine instructions that may interleave, possibly resulting in a lost update to x.

```
int x;                  cilk void foo1()
Cilk_lockvar A;         {
                          Cilk_lock(A);
cilk int main()           x += 2;
{                         Cilk_unlock(A);
  x = 2;                }
  Cilk_lock_init(A);
  spawn foo1();         cilk void foo2()
  spawn foo2();         {
  printf("%d", x);        Cilk_lock(A);
  return 0;               x *= 3;
}                         Cilk_unlock(A);
                        }
```

**Figure 1-3**: A Cilk program that incorporates user-level locking to produce atomic critical sections. Locks are declared as `Cilk_lockvar` variables, and must be initialized by `Cilk_lock_init()` statements. The function `Cilk_lock()` acquires a specified lock, and `Cilk_unlock()` releases a lock.

Figure 1-2 shows an example of this interleaving occurring. Processor 1 performs the `x += 2` operation at the "same time" as processor 2 performs the `x *= 3` operation. The individual machine instructions that comprise these operations interleave, producing the value 6 as the final value of `x`.

This behavior is likely to be a bug, but it may be the programmer's intention. It is also possible that the programmer intended 8 or 12 to be legal final values for `x`, but not 4 or 6. This behavior could be legitimately achieved through the use of mutual-exclusion locks. A ***lock*** is a language construct, typically implemented as a location in shared memory, that can be acquired and released but that is guaranteed to be acquired by at most one thread at once. In other words, locks allow the programmer to force certain sections of the code, called ***critical sections***, to be "atomic" with respect to each other. Two operations are ***atomic*** if the instructions that comprise them cannot be interleaved. Figure 1-3 shows the program in Figure 1-1 with locks added. In this version, the value of `x` printed by `main` may be either 8 or 12, but cannot be 4 or 6.

The program in Figure 1-3 is nondeterministic, but it is somehow "less nondeterministic" than the program in Figure 1-1. Indeed, while Figure 1-3 uses locks to "control" nondeterminism, the locks themselves are inherently nondeterministic,

```
int x;                  cilk void foo1()
Cilk_lockvar A;         {
Cilk_lockvar B;           Cilk_lock(A);
                          x += 2;
cilk int main()           Cilk_unlock(A);
{                       }
  x = 2;
  Cilk_lock_init(A);    cilk void foo2()
  Cilk_lock_init(B);    {
  spawn foo1();           Cilk_lock(B);
  spawn foo2();           x *= 3;
  printf("%d", x);        Cilk_unlock(B);
  return 0;             }
}
```

**Figure 1-4**: A Cilk program that uses locks but that still contains a data race. The distinct locks `A` and `B` do not prevent the updates to `x` from interleaving.

because the semantics of locks is that any of the threads trying to acquire a lock may in fact be the one to get it. In fact, it is arguable that any Cilk program is nondeterministic, because memory updates happen in different orders depending on scheduling.

Rather than attempt to discuss these issues with such ambiguity, we present a formal model for defining nondeterminism. Under this model, we can precisely define multiple kinds of nondeterminism. In particular, we define the concept of a ***data race***: intuitively, a situation where parallel threads could update (or update and access) a memory location "simultaneously." Figure 1-1 contains a data race, whereas Figure 1-3 does not. It should be noted, however, that the mere presence of locks does not preclude data races. It still necessary to use the right locks in the right places. Figure 1-4 shows an example where locks have been used (presumably) incorrectly. The two distinct locks `A` and `B` do not have any effect on each other, so the updates to `x` once again may interleave in a data race.

Data races may not exactly represent the form of nondeterminism that programmers care about. Data races are *likely* to be bugs, however, and they are interesting because they are a form of nondeterminism that we can hope to detect automatically. By automatically detecting data races, we can provide debugging information to the

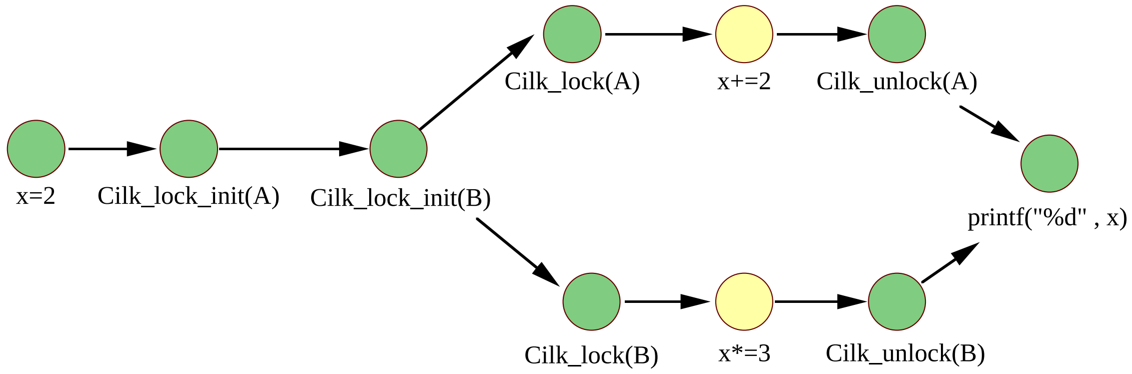


**Figure 1-5**: The computation dag for the program in Figure 1-3. A dag race exists between the two highlighted nodes.

programmer that is of great use when trying to track down nondeterminacy bugs.

Unfortunately, even detection of data races is computationally too difficult to be done in a practical debugging tool. Instead, the Nondeterminator-2 detects "dag races." Roughly, a dag race race is like a data race, but the question of whether two memory accesses could occur simultaneously is approximated. We say that an execution of a Cilk program generates a ***computation***, which is a directed acyclic graph (dag) where the nodes represent the instructions of the program and the edges represent the parallel control constructs. The dag for Figure 1-4 is shown in Figure 1-5.[1] The dag is an approximation of other possible executions of the same program on the same input; that is, we consider the possible executions of the program on that input to be the topological sorts of the dag in which each lock is held at most once at any given time. A ***dag race***, then, occurs when two instructions that are unrelated in the dag both access the same memory location, at least one of the accesses is a write, and no common lock is held across both of the accesses. Figure 1-5 has a dag race between the two highlighted nodes.

As we shall see, the dag races that the Nondeterminator-2 detects are not the

[1] This picture of the dag is a simplification; a formal method for construction of the dag is given in Chapter 7.

same thing as data races. Nonetheless, experience shows that dag races are a good enough approximation to be useful to report as debugging information to the programmer. Furthermore, we show that for some set of programs, the dag races precisely correspond to data races. One such set of programs is the "abelian" programs in which all critical sections protected by the same lock "commute": intuitively, the critical sections produce the same effect regardless of scheduling. We show that if a (deadlock-free) abelian program generates a computation with no dag races, then the program is ***determinate***: all schedulings produce the same final result. The consequence, therefore, is that for an abelian program, the Nondeterminator-2 can verify the determinacy of the program on a given input.

The Nondeterminator-2 cannot provide such a guarantee for nonabelian programs. Even for such programs, however, we expect that reporting dag races to the user provides a useful debugging heuristic. Indeed, this approach has been implicitly taken by all previous dynamic race-detection tools.

## Race-detection algorithms

In previous work, some efforts have been made to detect data races statically (at compile-time) [31, 42]. Static debuggers have the advantage that they sometimes can draw conclusions about the program for all inputs. Since understanding any nontrivial semantics of the program is generally undecidable, however, most race detectors are dynamic tools in which potential races are detected at runtime by executing the program on a given input. Some dynamic race detectors perform a post-mortem analysis based on program execution traces [12, 21, 29, 32], while others perform an "on-the-fly" analysis during program execution. On-the-fly debuggers directly instrument memory accesses via the compiler [10, 11, 14, 15, 28, 36], by binary rewriting [39], or by augmenting the machine's cache coherence protocol [30, 37].

In this thesis, we present two race detection algorithms which are based on the Nondeterminator [14], a tool that finds dag races in Cilk programs that do not use locks. The Nondeterminator executes a Cilk program serially on a given input, maintaining an efficient "SP-bags" data structure to keep track of the logical series/parallel relationships between threads. For a Cilk program that runs serially in time $T$ and accesses $V$ shared-memory locations, the Nondeterminator runs in $O(T\,\alpha(V,V))$ time and $O(V)$ space, where $\alpha$ is Tarjan's functional inverse of Ackermann's function, which for all practical purposes is at most 4.

The Nondeterminator-2, the tool presented here, finds dag races in Cilk programs that use locks. This race detector contains two algorithms, both of which use the same efficient SP-bags data structure from the original Nondeterminator. The first of these algorithms, All-Sets, is an on-the-fly algorithm that, like most other race-detection algorithms, assumes that no locks are held across parallel control statements, such as `spawn` and `sync`. The second algorithm, Brelly, is a faster on-the-fly algorithm, but in addition to reporting dag races as bugs, it also reports as bugs some complex locking protocols that are probably undesirable but that may be race free.

The All-Sets algorithm executes a Cilk program serially on a given input and either detects a dag race in the computation or guarantees that none exist. For a Cilk program that runs serially in time $T$, accesses $V$ shared-memory locations, uses a total of $n$ locks, and holds at most $k \ll n$ locks simultaneously, All-Sets runs in $O(n^k T\,\alpha(V,V))$ time and $O(n^k V)$ space. Tighter, more complicated bounds on All-Sets are given in Chapter 2.

In previous work, Dinning and Schonberg's "Lock Covers" algorithm [11] also detects all dag races in a computation. The All-Sets algorithm improves the Lock Covers algorithm by generalizing the data structures and techniques from the original Nondeterminator to produce better time and space bounds. Perkovic and Keleher [37] offer an on-the-fly race-detection algorithm that "piggybacks" on a cache-coherence protocol for lazy release consistency. Their approach is fast (about twice the serial work, and the tool runs in parallel), but it only catches races that actually occur during a parallel execution, not those that are logically present in the computation.

Although the asymptotic performance bounds of All-Sets are the best to date, they are a factor of $n^k$ larger in the worst case than those for the original Nondeterminator. The Brelly algorithm is asymptotically faster than All-Sets, and its performance bounds are only a factor of $k$ larger than those for the original Nondeterminator. For a Cilk program that runs serially in time $T$, accesses $V$ shared-memory locations, and holds at most $k$ locks simultaneously, the serial Brelly algorithm runs in $O(kT\,\alpha(V,V))$ time and $O(kV)$ space. Since most programs do not hold many locks simultaneously, this algorithm runs in nearly linear time and space. The improved performance bounds come at a cost, however. Rather than detecting dag races directly, Brelly only detects violations of a "locking discipline" that precludes dag races.

A ***locking discipline*** is a programming methodology that dictates a restriction on the use of locks. For example, many programs adopt the discipline of acquiring locks in a fixed order so as to avoid deadlock [22]. Similarly, the "umbrella" locking discipline precludes dag races by requiring that each location be protected by the same lock within every parallel subcomputation of the computation. Threads that are in series may use different locks for the same location (or possibly even none, if no parallel accesses occur), but if two threads in series are both in parallel with a third and all access the same location, then all three threads must agree on a single lock for that location. If a program obeys the umbrella discipline, a dag race cannot occur, because parallel accesses are always protected by the same lock. The Brelly algorithm detects violations of the umbrella locking discipline.

Savage et al. [39] originally suggested that efficient debugging tools can be developed by requiring programs to obey a locking discipline. Their Eraser tool enforces a simple discipline in which any shared variable is protected by a single lock throughout the course of the program execution. Whenever a thread accesses a shared variable, it must acquire the designated lock. This discipline precludes dag races from occurring, and Eraser finds violations of the discipline in $O(kT)$ time and $O(kV)$ space. (These bounds are for the serial work; Eraser actually runs in parallel.) Eraser only works in a parallel environment containing several linear threads, however, with no nested parallelism or thread joining as is permitted in Cilk. In addition, since Eraser does not recognize the series/parallel relationship of threads, it does not properly understand at what times a variable is actually shared. Specifically, it heuristically guesses when the "initialization phase" of a variable ends and the "sharing phase" begins, and thus it may miss some dag races.

In comparison, our Brelly algorithm performs nearly as efficiently, is guaranteed to find all violations, and importantly, supports a more flexible discipline. In particular, the umbrella discipline allows separate program modules to be composed in series without agreement on a global lock for each location. For example, an application may have three phases—an initialization phase, a work phase, and a clean-up phase—which can be developed independently without agreeing globally on the locks used to protect locations. If a fourth module runs in parallel with all of these phases and accesses the same memory locations, however, the umbrella discipline does require that all phases agree on the lock for each shared location. Thus, although the umbrella discipline is more flexible than Eraser's discipline, it is more restrictive than what a general dag-race detection algorithm, such as All-Sets, permits.

Figure 1-6 compares the asymptotic performance of All-Sets and Brelly with other race detection algorithms in the literature. A more in-depth discussion of this comparison is given in Chapter 4.

## Using the Nondeterminator-2

In addition to presenting the All-Sets and Brelly algorithms themselves, we discuss practical issues surrounding their use. Specifically, we explain how they can be used when memory is allocated and freed dynamically. We describe techniques for annotating code in order to make dag race reports more useful for practical debugging purposes. Additionally, we present timings of the algorithms on a few example Cilk programs.

Finally, we present an in-depth case study of our experiences parallelizing a large radiosity application. Radiosity is a graphics algorithm for modeling light in diffuse environments. Figure 1-7 shows a scene in which radiosity was used to model the reflections of light off of the walls of a maze. The majority of the calculation time for radiosity is spent calculating certain properties of the scene geometry. These calculations can be parallelized, and Cilk is ideally suited for this parallelization, because its load-balancing scheduler is provably good, and so can obtain speedup

| Algorithm | Handles locks | Handles series-parallel programs | Detects | Time per memory access | Total space |
|---|---|---|---|---|---|
| English-Hebrew labeling [36] | NO | YES | Dag races | $O(pt)$ | $O(Vt + \min(bp, Vtp))$ |
| Task Recycling [10] | NO | YES | Dag races | $O(t)$ | $O(t^2 + Vt)$ |
| Offset-span Labeling [28] | NO | YES | Dag races | $O(p)$ | $O(V + \min(bp, Vp))$ |
| SP-Bags [14] | NO | YES | Dag races | $O(\alpha(V, V))$ | $O(V)$ |
| Lock Covers [11] | YES | YES | Dag races | $O(tn^k)$ | $O(t^2 + tn^k V)$ |
| Eraser [39] | YES | NO | Eraser discipline violations | $O(k)$ | $O(kV)$ |
| All-Sets | YES | YES | Dag races | $O(n^k \, \alpha(V, V)))$ | $O(n^k V)$ |
| Brelly | YES | YES | Umbrella discipline violations | $O(k \, \alpha(V, V))$ | $O(kV)$ |

$p$ = maximum depth of nested parallelism

$t$ = maximum number of logically concurrent threads

$V$ = number of shared memory locations used

$b$ = total number of threads in the computation

$k$ = maximum number of locks held simultaneously

**Figure 1-6**: Comparison of race detection algorithms. Tighter, more complicated bounds are given for All-Sets (and Lock Covers) in Figure 4-1.

even for such irregular calculations.

Parallelization speedup, however, is not particularly impressive if the same result could be achieved by optimizing the serial execution. Therefore, it is usually not desirable to rewrite applications for parallel execution, because the optimizations in the serial code might be lost. So instead of implementing our own radiosity code, we downloaded a large radiosity application developed at the Computer Graphics Research Group of the Katholieke Universiteit Leuven, in Belgium [2]. Since the code is written in C, and Cilk is a simple extension of C, running the code as a Cilk program is effortless.

**Figure 1-7**: A maze scene rendered after 100 iterations of the radiosity algorithm.

The difficulty, however, is that the program is large, consisting of 75 source files and 23,000 lines of code. The code was not written with parallelization in mind, so there are portions where memory is shared "unnecessarily." That is, operations that in principle could be independent actually write to the same memory locations. These conflicts need to be resolved if those operations are to be run in parallel. Searching through the code for such problems would be very tedious. The Nondeterminator-2, however, provides a much faster approach. We just run in parallel those operations that are "in principle" independent, and use the Nondeterminator-2 to find the places in the code where this parallelization failed. In this way, we are directly pointed to the problem areas of the code and have no need to examine pieces of the code that don't demonstrate any races. Our resulting Cilk radiosity code achieves a 5.97 times speedup on 8 processors.

## Organization of this thesis

This thesis is organized into three major parts.

Part I discusses the race-detection algorithms. Chapter 2 presents the All-Sets algorithm for detecting dag races in a Cilk computation, and Chapter 3 presents the Brelly algorithm for detecting umbrella discipline violations. Chapter 4 then gives a comparison of the asymptotic performance of these algorithms with other race-detection algorithms in the literature.

Part II presents our theory of nondeterminism. Chapter 5 presents a framework for defining nondeterminism, and data races in particular. Chapter 6 shows that precise detection of data races is computationally infeasible. Chapter 7 explains why the dag races that are detected by the Nondeterminator-2 are not the same thing as data races. Chapter 8, however, defines the notion of abelian programs, and shows that there is a provable correspondence between dag races and data races for abelian programs. Furthermore, that chapter shows that the Nondeterminator-2 can verify the determinacy of deadlock-free abelian programs. A complicated proof of one lemma needed for this result is left to Appendix A.

Finally, in Part III, we discuss some practical considerations surrounding the use of the Nondeterminator-2. Chapter 9 discusses how to detect races in the presence of dynamic memory allocation and how to reduce the number of "false race reports" that the Nondeterminator-2 produces. Timings of our implementation of the Nondeterminator-2 are also given in that chapter. Some of the ideas described in Chapter 9 were inspired by our experiences parallelizing the radiosity application; these experiences are described in Chapter 10. Chapter 11 offers some concluding remarks.

Some of the results in this thesis appear in [6] and represent joint work with Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, and Keith Randall.

# Part I

# Race-Detection Algorithms

# Chapter 2

# The All-Sets Algorithm

In this chapter, we present the All-Sets algorithm, which detects dag races in Cilk computations.[1] We first give some background on Cilk and explain the series-parallel structure of its computations. Next we review the SP-bags algorithm[14] used by the original Nondeterminator. We then we present the All-Sets algorithm itself, show that it is correct, and analyze its performance. Specifically, we that for a program that runs serially in time $T$, accesses $V$ shared memory locations, uses a total of $n$ locks, and holds at most $k \ll n$ locks simultaneously, All-Sets run in $O(n^k T\, \alpha(V, V))$ time and $O(n^k V)$ space, where $\alpha$ is Tarjan's functional inverse of Ackermann's function. Furthermore, All-Sets guarantees to find a dag race in the generated computation if and only if such a race exists.

## Cilk

Cilk is an algorithmic multithreaded language. The idea behind Cilk is to allow programmers to easily express the parallelism of their programs, and to have the runtime system take care of the details of running the program on many processors. Cilk's scheduler uses a work-stealing algorithm to achieve provably good performance. While this feature is not the main focus of this paper, it surfaces again as motivation for the radiosity example.

[1] Some of the results in this chapter appear in [6].

In order to make it as easy as possible for programmers to express parallelism, Cilk was designed as a simple extension to C. A Cilk program is a C program with a few keywords added. Furthermore, a Cilk program running on one processor has the same semantics as the C program that is left when those keywords are removed. Cilk does not require programmers to know a priori on how many processors their programs will run.

The Cilk keyword `spawn`, when immediately preceding a function call, declares that the function may be run in parallel. In other words, the parent function that spawned the child is allowed to continue executing at the same time as the child function executes. The parent may later issue a `sync` instruction, which means that the parent must wait until all the children it has spawned complete before continuing.[2] Any procedure that spawns other procedures or that itself is spawned must be declared with the type qualifier `cilk`.

Figure 2-1 gives an example Cilk procedure that computes the $n$th Fibonacci number. The two recursive cases of the Fibonacci calculation are spawned off in parallel. The code then syncs, which forces it to wait for the two spawned subcomputations to complete. Once they have done so, their results are available to be accumulated and returned.

Additionally, Cilk provides the user with mutual-exclusion locks. A lock is essentially a location in shared memory that can be "acquired" or "released." It is guaranteed, however, that at most one thread can acquire a given lock at once. The command `Cilk_lock()` acquires a specified lock, and `Cilk_unlock()` releases a specified lock. If the lock is already acquired then `Cilk_lock()` "spins," meaning that it waits until the lock is released, and then attempts to acquire it again. We assume in this thesis, as does the race-detection literature, that parallel control constructs are disallowed while locks are held.[3]

[2] The semantics of `spawn` and `sync` are similar to that of `fork/join`, but `spawn` and `sync` are lightweight operations.

[3] The Nondeterminator-2 can still be used with programs for which this assumption does not hold, but the race detector prints a warning, and some races may be missed. We are developing extensions of the Nondeterminator-2's detection algorithms that work properly for programs that hold locks across parallel control constructs. See [5] for more discussion.

```
cilk int fib(int n)
{
  int x;
  int y;

  if (n < 2)
    return n;

  x = spawn fib(n-1);
  y = spawn fib(n-1);
  sync;
  return (x+y);
}
```

**Figure 2-1**: A Cilk procedure that computes the $n$th Fibonacci number.

The computation of a Cilk program on a given input can be viewed as a directed acyclic graph, or ***dag***, in which vertices are instructions and edges denote ordering constraints imposed by control statements. A Cilk `spawn` statement generates a vertex with out-degree 2, and a Cilk `sync` statement generates a vertex whose in-degree is 1 plus the number of subprocedures syncing at that point.

We define a ***thread*** to be a maximal sequence of vertices that does not contain any parallel control constructs. If there is a path in the dag from thread $e_1$ to thread $e_2$, then we say that the threads are ***logically in series***, which we denote by $e_1 \prec e_2$. If there is no path in the dag between $e_1$ and $e_2$, then they are ***logically in parallel***, $e_1 \parallel e_2$. Only the series relation $\prec$ is transitive. A ***dag race*** exists on a Cilk computation if two threads $e_1 \parallel e_2$ access the same memory location while holding no locks in common, and at least one of the threads writes the location.

The computation dag generated by a Cilk program can itself be represented as a binary ***series-parallel parse tree***, as illustrated in Figure 2-2. In the parse tree of a Cilk computation, leaf nodes represent threads. Each internal node is either an ***S-node*** if the computation represented by its left subtree logically precedes the computation represented by its right subtree, or a ***P-node*** if its two subtrees' computations are logically in parallel.

A parse tree allows the series/parallel relation between two threads $e_1$ and $e_2$

```
int x;
Cilk_lockvar A, B;

cilk void foo1() {
  Cilk_lock(A);
  Cilk_lock(B);
  x += 5;
  Cilk_unlock(B);
  Cilk_unlock(A);
}

cilk void foo2() {
  Cilk_lock(A);
  x -= 3;
  Cilk_unlock(A);
}
```

```
cilk void foo3() {
  Cilk_lock(B);
  x++;
  Cilk_unlock(B);
}

cilk int main() {
  Cilk_lock_init(A);
  Cilk_lock_init(B);
  x = 0;
  spawn foo1();
  spawn foo2();
  spawn foo3();
  sync;
  printf("%d", x);
}
```

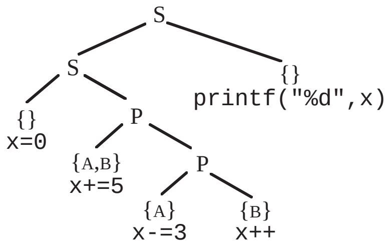


**Figure 2-2**: A Cilk program and the associated series-parallel parse tree, abbreviated to show only the accesses to shared location x. Each leaf is labeled with a code fragment that accesses x, with the set of locks held at that access shown above the code fragment.

to be determined by examining their least common ancestor, which we denote by LCA$(e_1, e_2)$. If LCA$(e_1, e_2)$ is a P-node, the two threads are logically in parallel ($e_1 \parallel e_2$). If LCA$(e_1, e_2)$ is an S-node, the two threads are logically in series: $e_1 \prec e_2$, assuming that $e_1$ precedes $e_2$ in a left-to-right depth-first tree walk of the parse tree.

## The original Nondeterminator

The original Nondeterminator uses the efficient SP-BAGS algorithm to detect dag races in Cilk programs that do not use locks. The SP-BAGS algorithm executes a Cilk program on a given input in serial, depth-first order. This execution order mirrors that of normal C programs: every subcomputation that is spawned executes completely before the procedure that spawned it continues. Every spawned procedure[4] is given a unique ID at runtime. These IDs are kept in the fast disjoint-set data structure [8, Chapter 22] analyzed by Tarjan [43]. The data structure maintains a dynamic collection $\Sigma$ of disjoint sets and provides three elementary operations:

**Make-Set**$(x)$**:** $\Sigma \leftarrow \Sigma \cup \{\{x\}\}$.

[4] Technically, by "procedure" we mean "procedure *instance*," that is, the runtime state of the procedure.

```
SPAWN procedure $F$:
    $S_F \leftarrow$ MAKE-SET($F$)
    $P_F \leftarrow \{\}$

SYNC in a procedure $F$:
    $S_F \leftarrow$ UNION($S_F, P_F$)
    $P_F \leftarrow \{\}$

RETURN from procedure $F'$ to $F$:
    $P_F \leftarrow$ UNION($P_F, S_{F'}$)
```

**Figure 2-3**: The SP-BAGS algorithm for updating S-bags and P-bags, which are represented as disjoint sets.

**Union**$(X, Y)$**:** $\Sigma \leftarrow \Sigma - \{X, Y\} \cup \{X \cup Y\}$. The sets $X$ and $Y$ are destroyed.

**Find-Set**$(x)$**:** Returns the set $X \in \Sigma$ such that $x \in X$.

Tarjan shows that any $m$ of these operations on $n$ sets take a total of $O(m\,\alpha(m, n))$ time.

During the execution of the SP-bags algorithm, two "bags" of procedure ID's are maintained for every Cilk procedure on the call stack. These bags have the following contents:

- The ***S-bag*** $S_F$ of a procedure $F$ contains the ID's of those descendants of $F$'s completed children that logically "precede" the currently executing thread, as well as the ID for $F$ itself.
- The ***P-bag*** $P_F$ of a procedure $F$ contains the ID's of those descendants of $F$'s completed children that operate logically "in parallel" with the currently executing thread.

The S-bags and P-bags are represented as sets using the disjoint-set data structure. At each parallel control construct of the program, the contents of the bags are updated as described in Figure 2-3. To determine the logical relationship of the currently executing thread with any already executed thread only requires a FIND-SET operation, which runs in amortized $\alpha(V, V)$ time. If the set found is an S-bag, the threads are in series, whereas if a P-bag is found, the threads are in parallel.

In addition, SP-BAGS maintains a ***shadow space*** that has an entry corresponding to each location of shared memory. For a location $l$ of shared memory, the corresponding shadow space entry keeps information about previous accesses to $l$. This information is used to find previous threads that have accessed the same location as the current thread.

## The All-Sets algorithm

The ALL-SETS algorithm is an extension of the SP-BAGS algorithm that detects dag races in Cilk programs that use locks. The ALL-SETS algorithm also uses S-bags and P-bags to determine the series/parallel relationship between threads. Its shadow space *lockers* is more complex than the shadow space of SP-BAGS, however, because it keeps track of which locks were held by previous accesses to the various locations.

The ***lock set*** of an access is the set of locks held by the thread when the access occurs. The ***lock set*** of several accesses is the intersection of their respective lock sets. If the lock set of two parallel accesses to the same location is empty, and at least one of the accesses is a WRITE, then a dag race exists. To simplify the description and analysis of the race detection algorithm, we shall use a small trick to avoid the extra condition for a race that "at least one of the accesses is a WRITE." The idea is to introduce a ***fake lock*** for read accesses called the R-LOCK, which is implicitly acquired immediately before a READ and released immediately afterwards. The fake lock behaves from the race detector's point of view just like a normal lock, but during an actual computation, it is never actually acquired and released (since it does not actually exist). The use of R-LOCK simplifies the description and analysis of ALL-SETS, because it allows us to state the condition for a dag race more succinctly: *if the lock set of two parallel accesses to the same location is empty, then a dag race exists.* By this condition, a dag race (correctly) does not exist for two read accesses, since their lock set contains the R-LOCK.

The entry *lockers*[$l$] in ALL-SETS' shadow space stores a list of ***lockers***: threads that access location $l$, each paired with the lock set that was held during the access.

```
LOCK($A$)
    Add $A$ to $H$

UNLOCK($A$)
    Remove $A$ from $H$

ACCESS($l$) in thread $e$ with lock set $H$
    for each $\langle e', H' \rangle \in lockers[l]$
        do if $e' \parallel e$ and $H' \cap H = \{\}$
            then declare a dag race
    $redundant \leftarrow$ FALSE
    for each $\langle e', H' \rangle \in lockers[l]$
        do if $e' \prec e$ and $H' \supseteq H$
            then $lockers[l] \leftarrow lockers[l] - \{\langle e', H' \rangle\}$
           if $e' \parallel e$ and $H' \subseteq H$
            then $redundant \leftarrow$ TRUE
    if $redundant =$ FALSE
      then $lockers[l] \leftarrow lockers[l] \cup \{\langle e, H \rangle\}$
```

**Figure 2-4**: The ALL-SETS algorithm. The operations for the `spawn`, `sync`, and `return` actions are unchanged from the SP-BAGS algorithm.

If $\langle e, H \rangle \in lockers[l]$, then thread $e$ accesses location $l$ while holding the lock set $H$.

location $l$ is accessed by thread $e$ while it holds the lock set $H$.

As an example of what the shadow space *lockers* may contain, consider a thread $e$ that performs the following:

```
Cilk_lock(A); Cilk_lock(B);
READ(l)
Cilk_unlock(B); Cilk_unlock(A);
Cilk_lock(B); Cilk_lock(C);
WRITE(l)
Cilk_unlock(C); Cilk_unlock(B);
```

For this example, the list $lockers[l]$ contains two lockers—$\langle e, \{\texttt{A}, \texttt{B}, \text{R-LOCK}\} \rangle$ and $\langle e, \{\texttt{B}, \texttt{C}\} \rangle$.

The ALL-SETS algorithm is shown in Figure 2-4. Intuitively, this algorithm records all lockers, but it is careful to prune redundant lockers, keeping at most one locker per distinct lock set. Locks are added and removed from the global lock set $H$ at `Cilk_lock` and `Cilk_unlock` statements. Lines 1–3 check to see if a dag race has occurred and report any violations. Lines 5–11 then add the current locker to the *lockers* shadow space and prune redundant lockers.

## Correctness of All-Sets

Before proving the correctness of All-Sets, we restate two lemmas from [14].

**Lemma 1** *Suppose that three threads $e_1$, $e_2$, and $e_3$ execute in order in a serial, depth-first execution of a Cilk program, and suppose that $e_1 \prec e_2$ and $e_1 \parallel e_3$. Then, we have $e_2 \parallel e_3$.* ■

**Lemma 2 (Pseudotransitivity of $\parallel$)** *Suppose that three threads $e_1$, $e_2$, and $e_3$ execute in order in a serial, depth-first execution of a Cilk program, and suppose that $e_1 \parallel e_2$ and $e_2 \parallel e_3$. Then, we have $e_1 \parallel e_3$.* ■

We now prove that the All-Sets algorithm is correct.

**Theorem 3** *The* All-Sets *algorithm detects a dag race in a computation of a Cilk program running on a given input if and only if a dag race exists in the computation.*

*Proof:* (⇒) To prove that any race reported by the All-Sets algorithm really exists in the computation, observe that every locker added to *lockers*$[l]$ in line 11 consists of a thread and the lock set held by that thread when it accesses $l$. The algorithm declares a race when it detects in line 2 that the lock set of two parallel accesses (by the current thread $e$ and one from *lockers*$[l]$) is empty, which is exactly the condition required for a dag race.

(⇐) Assuming a dag race exists in a computation, we shall show that a dag race is reported. If a dag race exists, then we can choose two threads $e_1$ and $e_2$ such that $e_1$ is the last thread before $e_2$ in the serial execution that has a dag race with $e_2$. If we let $H_1$ and $H_2$ be the lock sets held by $e_1$ and $e_2$, respectively, then we have $e_1 \parallel e_2$ and $H_1 \cap H_2 = \{\}$.

We first show that immediately after $e_1$ executes, *lockers*$[l]$ contains some thread $e_3$ that races with $e_2$. If $\langle e_1, H_1 \rangle$ is added to *lockers*$[l]$ in line 11, then $e_1$ is such an $e_3$. Otherwise, the *redundant* flag must have been set in line 9, so there must exist a locker $\langle e_3, H_3 \rangle \in$ *lockers*$[l]$ with $e_3 \parallel e_1$ and $H_3 \subseteq H_1$. Thus, by pseudotransitivity (Lemma 2), we have $e_3 \parallel e_2$. Moreover, since $H_3 \subseteq H_1$ and $H_1 \cap H_2 = \{\}$, we have $H_3 \cap H_2 = \{\}$, and therefore $e_3$, which belongs to *lockers*$[l]$, races with $e_2$.

To complete the proof, we now show that the locker $\langle e_3, H_3 \rangle$ is not removed from *lockers*$[l]$ between the times that $e_1$ and $e_2$ are executed. Suppose to the contrary that $\langle e_4, H_4 \rangle$ is a locker that causes $\langle e_3, H_3 \rangle$ to be removed from *lockers*$[l]$ in line 7. Then, we must have $e_3 \prec e_4$ and $H_3 \supseteq H_4$, and by Lemma 1, we have $e_4 \parallel e_2$. Moreover, since $H_3 \supseteq H_4$ and $H_3 \cap H_2 = \{\}$, we have $H_4 \cap H_2 = \{\}$, contradicting the choice of $e_1$ as the last thread before $e_2$ to race with $e_2$.

Therefore, thread $e_3$, which races with $e_2$, still belongs to *lockers*$[l]$ when $e_2$ executes, and so lines 1–3 report a race. ∎

## Analysis of All-Sets

In Chapter 1, we claimed that for a Cilk program that executes in time $T$ on one processor, references $V$ shared memory locations, uses a total of $n$ locks, and holds at most $k \ll n$ locks simultaneously, the ALL-SETS algorithm can check this computation for dag races in $O(n^k T \,\alpha(V,V))$ time and using $O(n^k V)$ space. These bounds, which are correct but weak, are improved by the next theorem.

**Theorem 4** *Consider a Cilk program that executes in time $T$ on one processor, references $V$ shared memory locations, uses a total of $n$ locks, and holds at most $k$ locks simultaneously. The* ALL-SETS *algorithm checks this computation for dag races in $O(TL(k + \alpha(V,V)))$ time and $O(kLV)$ space, where $L$ is the maximum of the number of distinct lock sets used to access any particular location.*

*Proof:* First, observe that no two lockers in *lockers* have the same lock set, because the logic in lines 5–11 ensure that if $H = H'$, then locker $\langle e, H \rangle$ either replaces $\langle e', H' \rangle$

(line 7) or is considered redundant (line 9). Thus, there are at most $L$ lockers in the list *lockers*$[l]$. Each lock set takes at most $O(k)$ space, so the space needed for *lockers* is $O(kLV)$. The length of the list *lockers*$[l]$ determines the number of series/parallel relations that are tested. In the worst case, we need to perform $2L$ such tests (lines 2 and 6) and $2L$ set operations (lines 2, 6, and 8) per access. Each series/parallel test takes amortized $O(\alpha(V,V))$ time, and each set operation takes $O(k)$ time. Therefore, the ALL-SETS algorithm runs in $O(TL(k+\alpha(V,V)))$ time. ■

The looser bounds claimed in Chapter 1 of $O(n^k T\,\alpha(V,V))$ time and $O(n^k V)$ space for $k \ll n$ follow because $L \leq \sum_{i=0}^{k} \binom{n}{i} = O(n^k/k!)$. As we shall see in Chapter 9, however, we rarely see the worst-case behavior given by the bounds in Theorem 4.

# Chapter 3

# The Brelly Algorithm

In this section, we formally define the "umbrella locking discipline" and present the BRELLY algorithm for detecting violations of this discipline.[1] We prove that the BRELLY algorithm is correct and analyze its performance, which we show to be asymptotically better than that of ALL-SETS. Specifically, we show that for a program that runs serially in time $T$, accesses $V$ shared memory locations, uses a total of $n$ locks, and holds at most $k \ll n$ locks simultaneously, BRELLY runs in $O(kT\,\alpha(V,V))$ time using $O(kV)$ space, where $\alpha$ is Tarjan's functional inverse of Ackermann's function. We further prove that BRELLY guarantees to find a violation of the umbrella discipline in the computation if and only if a violation exists.

## The umbrella discipline

The umbrella discipline can be defined precisely in terms of the parse tree of a given Cilk computation. An ***umbrella*** of accesses to a location $l$ is a subtree rooted at a P-node containing accesses to $l$ in both its left and right subtrees, as is illustrated in Figure 3-1. An umbrella of accesses to $l$ is ***protected*** if its accesses have a nonempty lock set and ***unprotected*** otherwise. A program obeys the ***umbrella locking discipline*** if it contains no unprotected umbrellas. In other words, within each umbrella of accesses to a location $l$, all threads must agree on at least one lock to protect their

[1] Some of the results in this chapter appear in [6].

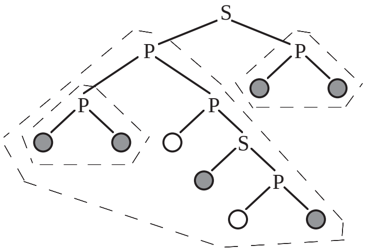


**Figure 3-1**: Three umbrellas of accesses to a location $l$. In this parse tree, each shaded leaf represents a thread that accesses $l$. Each umbrella of accesses to $l$ is enclosed by a dashed line.

accesses to $l$.

The next theorem shows that adherence to the umbrella discipline precludes dag races from occurring.

**Theorem 5** *A Cilk computation with a dag race violates the umbrella discipline.*

*Proof:* Any two threads involved in a dag race must have a P-node as their least common ancestor in the parse tree, because they operate in parallel. This P-node roots an unprotected umbrella, since both threads access the same location and the lock sets of the two threads are disjoint. ■

The umbrella discipline can also be violated by unusual, but dag-race free, locking protocols. For instance, suppose that a location is protected by three locks and that every thread always acquires two of the three locks before accessing the location. No single lock protects the location, but every pair of such accesses is mutually exclusive. The ALL-SETS algorithm properly certifies this bizarre example as race-free, whereas BRELLY detects a discipline violation. In return for disallowing these unusual locking protocols (which in any event are of dubious value), BRELLY checks programs asymptotically faster than ALL-SETS.

## The Brelly algorithm

Like ALL-SETS, the BRELLY algorithm extends the SP-BAGS algorithm used in the original Nondeterminator and uses the R-LOCK fake lock for read accesses (see Chapter 2). Figure 3-2 gives pseudocode for BRELLY. Like the SP-BAGS algorithm,

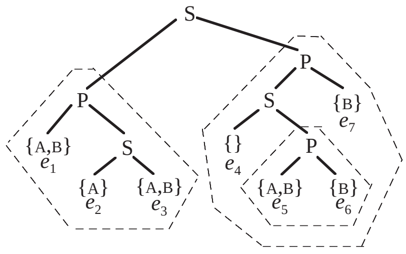
```
LOCK($A$)
    Add $A$ to $H$

UNLOCK($A$)
    Remove $A$ from $H$

ACCESS($l$) in thread $e$ with lock set $H$
if $accessor[l] \prec e$
  then ▷ serial access
       $locks[l] \leftarrow H$, leaving $nonlocker[h]$ with its old
         nonlocker if it was already in $locks[l]$ but
         setting $nonlocker[h] \leftarrow accessor[l]$ otherwise
       for each lock $h \in locks[l]$
           do $alive[h] \leftarrow$ TRUE
       $accessor[l] \leftarrow e$
  else ▷ parallel access
       for each lock $h \in locks[l] - H$
           do if $alive[h] =$ TRUE
                then $alive[h] \leftarrow$ FALSE
                     $nonlocker[h] \leftarrow e$
       for each lock $h \in locks[l] \cap H$
           do if $alive[h] =$ TRUE and $nonlocker[h] \parallel e$
                then $alive[h] \leftarrow$ FALSE
       if no locks in $locks[l]$ are alive (or $locks[l] = \{\}$)
         then report violation on $l$ involving
                  $e$ and $accessor[l]$
              for each lock $h \in H \cap locks[l]$
                  do report access to $l$ without $h$
                       by $nonlocker[h]$
```

**Figure 3-2**: The BRELLY algorithm. While executing a Cilk program in serial depth-first order, at each access to a shared-memory location $l$, the code for ACCESS($l$) is executed. Locks are added and removed from the lock set $H$ at `Cilk_lock` and `Cilk_unlock` statements. To determine whether the currently executing thread is in series or parallel with previously executed threads, BRELLY uses the SP-bags data structure.

Brelly executes the program on a given input in serial depth-first order, maintaining the SP-bags data structure so that the series/parallel relationship between the currently executing thread and any previously executed thread can be determined quickly. Like the All-Sets algorithm, Brelly also maintains a set $H$ of currently held locks. In addition, Brelly maintains two shadow spaces of shared memory: *accessor*, which stores for each location the thread that performed the last "serial access" to that location; and *locks*, which stores the lock set of that access. Each entry in the *accessor* space is initialized to the initial thread (which logically precedes all threads in the computation), and each entry in the *locks* space is initialized to the empty set.

Unlike the All-Sets algorithm, Brelly keeps only a single lock set, rather than a list of lock sets, for each shared-memory location. For a location $l$, each lock in $locks[l]$ potentially belongs to the lock set of the largest umbrella of accesses to $l$ that includes the current thread. The Brelly algorithm tags each lock $h \in locks[l]$ with two pieces of information: a thread $nonlocker[h]$ and a flag $alive[h]$. The thread $nonlocker[h]$ is a thread that accesses $l$ without holding $h$. The flag $alive[h]$ indicates whether $h$ should still be considered to potentially belong to the lock set of the umbrella. To allow reports of violations to be more precise, the algorithm "kills" a lock $h$ by setting $alive[h] \leftarrow \text{FALSE}$ when it determines that $h$ does not belong to the lock set of the umbrella, rather than simply removing it from $locks[l]$.

Whenever Brelly encounters an access by a thread $e$ to a location $l$, it checks for a violation with previous accesses to $l$, updating the shadow spaces appropriately for future reference. If $accessor[l] \prec e$, we say the access is a ***serial access***, and the algorithm performs lines 2–5, setting $locks[l] \leftarrow H$ and $accessor[l] \leftarrow e$, as well as updating $nonlocker[h]$ and $alive[h]$ appropriately for each $h \in H$. If $accessor[l] \parallel e$, we say the access is a ***parallel access***, and the algorithm performs lines 6–17, killing the locks in $locks[l]$ that do not belong to the current lock set $H$ (lines 7–10) or whose nonlockers are in parallel with the current thread (lines 11–13). If Brelly finds in line 14 that there are no locks left alive in $locks[l]$ after a parallel access, it has found an unprotected umbrella, and it then reports a discipline violation in lines 15–17.



| thread | *accessor*$[l]$ | *locks*$[l]$ | access type |
|---|---|---|---|
| initial | $e_0$ | $\{\,\}$ | |
| $e_1$ | $e_1$ | $\{\text{A}(e_0), \text{B}(e_0)\}$ | serial |
| $e_2$ | $e_1$ | $\{\text{A}(e_0), \underline{\text{B}}(e_2)\}$ | parallel |
| $e_3$ | $e_1$ | $\{\text{A}(e_0), \underline{\text{B}}(e_2)\}$ | parallel |
| $e_4$ | $e_4$ | $\{\,\}$ | serial |
| $e_5$ | $e_5$ | $\{\text{A}(e_4), \text{B}(e_4)\}$ | serial |
| $e_6$ | $e_5$ | $\{\underline{\text{A}}(e_6), \text{B}(e_4)\}$ | parallel |
| $e_7$ | $e_5$ | $\{\underline{\text{A}}(e_6), \underline{\text{B}}(e_4)\}$ | parallel |

**Figure 3-3**: A sample execution of the Brelly algorithm. We restrict our attention to the algorithm's operation on a single location $l$. In the parse tree, each leaf represents an access to $l$ and is labeled with the thread that performs the access (e.g., $e_1$) and the lock set of that access (e.g., $\{\text{A}, \text{B}\}$). Umbrellas are enclosed by dashed lines. The table displays the values of *accessor*$[l]$ and *locks*$[l]$ after each thread's access. The nonlocker for each lock is given in parentheses after the lock, and killed locks are underlined. The "access type" column indicates whether the access is a parallel or serial access. A discipline violation is reported after the execution of $e_7$, because $e_7$ is a parallel access and no locks are left alive in *locks*$[l]$.

When reporting a violation, Brelly specifies the location $l$, the current thread $e$, and the thread *accessor*$[l]$. It may be that $e$ and *accessor*$[l]$ hold locks in common, in which case the algorithm uses the nonlocker information in lines 16–17 to report threads that accessed $l$ without each of these locks. Thus, every violation message printed by the algorithm always describes enough information to show that the umbrella in question is in fact unprotected.

Figure 3-3 illustrates how Brelly works. The umbrella containing threads $e_1$, $e_2$, and $e_3$ is protected by lock A but not by lock B, which is reflected in *locks*$[l]$ after thread $e_3$ executes. The umbrella containing $e_5$ and $e_6$ is protected by B but not by A, which is reflected in *locks*$[l]$ after thread $e_6$ executes. During the execution of thread $e_6$, A is killed and *nonlocker*[A] is set to $e_6$, according to the logic in lines 7–10. When

$e_7$ executes, B remains as the only lock alive in *locks*$[l]$ and *nonlocker*$[\text{B}]$ is $e_4$ (due to line 2 during $e_5$'s execution). Since $e_4 \parallel e_7$, lines 11–13 kill B, leaving no locks alive in *locks*$[l]$, properly reflecting the fact that no lock protects the umbrella containing threads $e_4$ through $e_7$. Consequently, the test in line 14 causes BRELLY to declare a violation at this point.

## Correctness of Brelly

The following two lemma will be helpful in proving the correctness of BRELLY.

**Lemma 6** *Suppose a thread $e$ performs a serial access to location $l$ during an execution of* BRELLY. *Then all previously executed accesses to $l$ logically precede $e$ in the computation.* ■

*Proof:* By transitivity of the $\prec$ relation, all serial accesses to $l$ that execute before $e$ logically precede $e$. We must also show the same for all parallel accesses to $l$ that are executed before $e$. Now, consider a thread $e'$ that performs a parallel access to $l$ before $e$ executes, and let $e'' \parallel e'$ be the thread stored in *accessor*$[l]$ when $e'$ executes its parallel access. Since $e''$ is a serial access to $l$ that executes before $e$, we have $e'' \prec e$. Consequently, we must have $e' \prec e$, because otherwise, by pseudotransitivity (Lemma 2) we would have $e'' \parallel e$, a contradiction. ■

**Lemma 7** *The* BRELLY *algorithm maintains the invariant that for any location $l$ and lock $h \in$ locks$[l]$, the thread* nonlocker$[h]$ *is either the initial thread or a thread that accessed $l$ without holding $h$.* ■

*Proof:* There are two cases in which *nonlocker*$[h]$ is updated. The first is in the assignment *nonlocker*$[h] \leftarrow e$ in line 10. This update only occurs when the current thread $e$ does not hold lock $h$ (line 7). The second case is when a lock's *nonlocker*$[h]$ is set to *accessor*$[l]$ in line 2. If this update occurs during the first access to $l$ in the program, then *accessor*$[l]$ is the initial thread. Otherwise, *locks*$[l]$ is the set of locks held during an access to $l$ in *accessor*$[l]$, since *locks*$[l]$ and *accessor*$[l]$ are updated together to the current lock set $H$ and current thread $e$, respectively, during a serial access (lines 2–5), and neither is updated anywhere else. Thus, if $h \notin$ *locks*$[l]$, which is the case if *nonlocker*$[h]$ is being set to *accessor*$[l]$ in line 2, then *accessor*$[l]$ did not hold lock $h$ during its access to $l$. ■

**Theorem 8** *The* Brelly *algorithm detects a violation of the umbrella discipline in a computation of a Cilk program running on a given input if and only if a violation exists.*

*Proof:* We first show that Brelly only detects actual violations of the discipline, and then we argue that no violations are missed. In this proof, we denote by *locks*$^*[l]$ the set of locks in *locks*$[l]$ that have TRUE *alive* flags.

($\Rightarrow$) Suppose that Brelly detects a violation caused by a thread $e$, and let $e_0 =$ *accessor*$[l]$ when $e$ executes. Since we have $e_0 \parallel e$, it follows that $p = \textsc{lca}(e_0, e)$ roots an umbrella of accesses to $l$, because $p$ is a P-node and it has an access to $l$ in both subtrees. We shall argue that the lock set $U$ of the umbrella rooted at $p$ is empty. Since Brelly only reports violations when *locks*$^*[l] = \{\}$, it suffices to show that $U \subseteq$ *locks*$^*[l]$ at all times after $e_0$ executes.

Since $e_0$ is a serial access, lines 2–5 cause *locks*$^*[l]$ to be the lock set of $e_0$. At this point, we know that $U \subseteq$ *locks*$^*[l]$, because $U$ can only contain locks held by every access in $p$'s subtree. Suppose that a lock $h$ is killed (and thus removed from *locks*$^*[l]$), either in line 9 or line 13, when some thread $e'$ executes a parallel access between the times that $e_0$ and $e$ execute. We shall show that in both cases $h \notin U$, and so $U \subseteq$ *locks*$^*[l]$ is maintained.

In the first case, if thread $e'$ kills $h$ in line 9, it does not hold $h$, and thus $h \notin U$.

In the second case, we shall show that $w$, the thread stored in *nonlocker*$[h]$ when $h$ is killed, is a descendant of $p$, which implies that $h \notin U$, because by Lemma 7, $w$ accesses $l$ without the lock $h$. Assume for the purpose of contradiction that $w$ is not a descendant of $p$. Then, we have $\textsc{lca}(w, e_0) = \textsc{lca}(w, e')$, which implies that $w \parallel e_0$, because $w \parallel e'$. Now, consider whether *nonlocker*$[h]$ was set to $w$ in line 10 or in line 2 (not counting when *nonlocker*$[h]$ is left with its old value in line 2). If line 10 sets *nonlocker*$[h] \leftarrow w$, then $w$ must execute before $e_0$, since otherwise, $w$ would be a parallel access, and lock $h$ would have been killed in line 9 by $w$ before $e'$ executes. By Lemma 6, we therefore have the contradiction that $w \prec e_0$. If line 2 sets *nonlocker*$[h] \leftarrow w$, then $w$ performs a serial access, which must be prior to the most recent serial access by $e_0$. By Lemma 6, we once again obtain the contradiction that $w \prec e_0$.

($\Leftarrow$) We now show that if a violation of the umbrella discipline exists, then BRELLY detects a violation. If a violation exists, then there must be an unprotected umbrella of accesses to a location $l$. Of these unprotected umbrellas, let $T$ be a maximal one in the sense that $T$ is not a subtree of another umbrella of accesses to $l$, and let $p$ be the P-node that roots $T$. The proof focuses on the values of *accessor*$[l]$ and *locks*$[l]$ just after $p$'s left subtree executes.

We first show that at this point, *accessor*$[l]$ is a left-descendant of $p$. Assume for the purpose of contradiction that *accessor*$[l]$ is not a left-descendant of $p$ (and is therefore not a descendant of $p$ at all), and let $p' = \text{LCA}(\textit{accessor}[l], p)$. We know that $p'$ must be a P-node, since otherwise *accessor*$[l]$ would have been overwritten in line 5 by the first access in $p$'s left subtree. But then $p'$ roots an umbrella that is a proper superset of $T$, contradicting the maximality of $T$.

Since *accessor*$[l]$ belongs to $p$'s left subtree, no access in $p$'s right subtree overwrites *locks*$[l]$, as they are all logically in parallel with *accessor*$[l]$. Therefore, the accesses in $p$'s right subtree may only kill locks in *locks*$[l]$. It suffices to show that by the time all accesses in $p$'s right subtree execute, all locks in *locks*$[l]$ (if any) have been killed, thus causing a race to be declared. Let $h$ be some lock in *locks*$^*[l]$ just after the left subtree of $p$ completes.

Since $T$ is unprotected, an access to $l$ unprotected by $h$ must exist in at least one of $p$'s two subtrees. If some access to $l$ is not protected by $h$ in $p$'s right subtree, then $h$ is killed in line 9. Otherwise, let $e_{left}$ be the most-recently executed thread in $p$'s left subtree that performs an access to $l$ not protected by $h$. Let $e'$ be the thread in *accessor*$[l]$ just after $e_{left}$ executes, and let $e_{right}$ be the first access to $l$ in the right subtree of $p$. We now show that in each of the following cases, we have $nonlocker[h] \parallel e_{right}$ when $e_{right}$ executes, and thus $h$ is killed in line 13.

Case 1: Thread $e_{left}$ is a serial access. Just after $e_{left}$ executes, we have $h \notin locks[l]$ (by the choice of $e_{left}$) and $accessor[l] = e_{left}$. Therefore, when $h$ is later placed in $locks[l]$ in line 2, $nonlocker[h]$ is set to $e_{left}$. Thus, we have $nonlocker[h] = e_{left} \parallel e_{right}$.

Case 2: Thread $e_{left}$ is a parallel access and $h \in locks[l]$ just before $e_{left}$ executes. Just after $e'$ executes, we have $h \in locks[l]$ and $alive[h] = \text{TRUE}$, since $h \in locks[l]$ when $e_{left}$ executes and all accesses to $l$ between $e'$ and $e_{left}$ are parallel and do not place locks into $locks[l]$. By pseudotransitivity (Lemma 2), $e' \parallel e_{left}$ and $e_{left} \parallel e_{right}$ implies $e' \parallel e_{right}$. Note that $e'$ must be a descendant of $p$, since if it were not, $T$ would be not be a maximal umbrella of accesses to $l$. Let $e''$ be the most recently executed thread before or equal to $e_{left}$ that kills $h$. In doing so, $e''$ sets $nonlocker[h] \leftarrow e''$ in line 10. Now, since both $e'$ and $e_{left}$ belong to $p$'s left subtree and $e''$ follows $e'$ in the execution order and comes before or is equal to $e_{left}$, it must be that $e''$ also belongs to $p$'s left subtree. Consequently, we have $nonlocker[h] = e'' \parallel e_{right}$.

Case 3: Thread $e_{left}$ is a parallel access and $h \notin locks[l]$ just before $e_{left}$ executes. When $h$ is later added to $locks[l]$, its $nonlocker[h]$ is set to $e'$. As above, by pseudotransitivity, $e' \parallel e_{left}$ and $e_{left} \parallel e_{right}$ implies $nonlocker[h] = e' \parallel e_{right}$.

In each of these cases, $nonlocker[h] \parallel e_{right}$ still holds when $e_{right}$ executes, since $e_{left}$, by assumption, is the most recent thread to access $l$ without $h$ in $p$'s left subtree. Thus, $h$ is killed in line 13 when $e_{right}$ executes. ■

## Analysis of Brelly

**Theorem 9** *On a Cilk program that executes serially in time $T$, uses $V$ shared-memory locations, and holds at most $k$ locks simultaneously, the* Brelly *algorithm runs in $O(kT\ \alpha(V,V))$ time and $O(kV)$ space.*

*Proof:* The total space is dominated by the *locks* shadow space. For any location $l$, the Brelly algorithm stores at most $k$ locks in $locks[l]$ at any time, since locks are placed in $locks[l]$ only in line 2 and $|H| \leq k$. Hence, the total space is $O(kV)$.

Each loop in Figure 3-2 takes $O(k)$ time if lock sets are kept in sorted order, excluding the checking of $nonlocker[h] \parallel e$ in line 12, which dominates the asymptotic running time of the algorithm. The total number of times $nonlocker[h] \parallel e$ is checked over the course of the program is at most $kT$, requiring $O(kT\,\alpha(V,V))$ time. ■

# Chapter 4

# Related Work

In this chapter, we compare the All-Sets and Brelly algorithms to previous race detection algorithms in the literature. Although they may not have made it explicit, these past algorithms also detect dag races, and not data races. (Further discussion of the difference between the two kinds of races is given in Chapters 5 and 7.) We focus on dynamic, on-the-fly debugging tools. On-the-fly tools detect races as the program executes and are generally more efficient than postmortem tools, which run detection algorithms on program execution traces.

Figure 4-1 summarizes the comparison of All-Sets and Brelly with previous work. (This figure is the figure from Chapter 1 with tighter bounds for All-Sets and Dinning and Schonberg's Lock Covers.) The All-Sets algorithm is the fastest algorithm that precisely detects dag races in programs that use locks. The Brelly algorithm is the fastest algorithm that detects locking discipline violations in fully series-parallel programs.

The original work in this area was the English-Hebrew labeling method proposed by Nudler and Rudolph [36]. Their model assumes nested parallelism similar to Cilk's `spawn/sync`, but does not address programs that use locks.[1] In order to determine the logical relation between threads, each thread is given an English label and a Hebrew label. The $i$th child of a thread is labeled with $i$ appended to its parents' label, where

[1] Nudler and Rudolph do discuss handling explicit synchronization operations between parallel threads, but we do not discuss that portion of the algorithm here.

| Algorithm | Handles locks | Handles series-parallel dags | Detects | Time per memory access | Total space |
|---|---|---|---|---|---|
| English-Hebrew labeling [36] | NO | YES | Dag races | $O(pt)$ | $O(Vt + \min(bp, Vtp))$ |
| Task Recycling [10] | NO | YES | Dag races | $O(t)$ | $O(t^2 + Vt)$ |
| Offset-span Labeling [28] | NO | YES | Dag races | $O(p)$ | $O(V + \min(bp, Vp))$ |
| SP-Bags [14] | NO | YES | Dag races | $O(\alpha(V, V))$ | $O(V)$ |
| Lock Covers [11] | YES | YES | Dag races | $O(tk^2L)$ | $O(t^2 + tkLV)$ |
| Eraser [39] | YES | NO | Eraser discipline violations | $O(k)$ | $O(kV)$ |
| All-Sets | YES | YES | Dag races | $O(L(k + \alpha(V, V)))$ | $O(kLV)$ |
| Brelly | YES | YES | Umbrella discipline violations | $O(k\,\alpha(V, V))$ | $O(kV)$ |

$p$ = maximum depth of nested parallelism
$t$ = maximum number of logically concurrent threads
$V$ = number of shared memory locations used
$b$ = total number of threads in the computation
$k$ = maximum number of locks held simultaneously
$L$ = maximum number of distinct lock sets used to access a location

**Figure 4-1**: Comparison of dag-race detection algorithms. This figure gives tighter bounds for All-Sets and Lock Covers than those given in Figure 1-6.

$i$ is counted left to right for the English label and right to left for the Hebrew label. If each label of $e_1$ is less than the corresponding label of $e_2$, then $e_1 \prec e_2$.

The length of the labels is $O(p)$, where $p$ is the maximum depth of nested parallelism. In addition to keeping the labels, the algorithm must keep an "access history" for each memory location. An ***access history*** is a list containing information on which threads have accessed the location. (The access histories are essentially designed to keep the same information that the Nondeterminator-2 keeps in its shadow spaces.) In this case, the access history is a list of pointers to labels. The access history for each location may grow as large as the maximum number of logically concurrent threads $t$. The reason for this potentially large growth is that all concurrent threads that read the location must be noted in the access history. The parallel relation $\parallel$ is not transitive. So if two threads $e_1 \parallel e_2$ both read a memory location, they must both be noted in the access history, because a write to that location by another thread $e_3$ must be checked with both $e_1$ and $e_2$ for dag races.

If the program uses a total of $V$ shared memory locations, then the algorithm keeps $O(Vt)$ pointers in access histories. With reference counting garbage collection, the storage for the labels can be bounded by $O(Vtp)$. This storage can also be bounded by $O(bp)$, where $b$ is the total number of threads in the execution. Thus, the total amount of space used by the English-Hebrew labeling scheme is $O(Vt+\min(bp, Vtp))$. At each memory access, the algorithm does $O(t)$ comparisons of size $O(p)$ labels, for a time of $O(pt)$. Essentially, then, the algorithm slows down ordinary execution by a factor of $O(pt)$.[2]

The task recycling algorithm, due to Dinning and Schonberg [10], records more information in order to reduce the time to check if two threads are concurrent. Like English-Hebrew labeling, the algorithm does not address programs with locks. The algorithm uses at most $t$ task identifiers, which it assigns to all the threads. To distinguish between multiple threads with the same task id, each thread is a given a unique version number for its task. In addition, each currently executing thread $e$ maintains a parent vector of size $t$. The $i$th entry in this vector denotes the largest version number for task $i$ that serially precedes $e$. Thus, determining the logical relationship between threads requires only a constant time operation—a vector lookup and version number comparison.

The task recycling algorithm, however, must still keep the $O(t)$ size access history for each memory location. Thus, at each access, the algorithm performs $O(t)$ operations, each taking $O(1)$ time, for a program slowdown of $O(t)$. Up to $t$ threads may

[2] To be fully precise, we should also mention the $O(p)$ time to create and join threads. This term can be ignored when compared with the $O(pt)$ operation at each memory access, and anyway memory accesses occur much more frequently than thread creation/termination in most programs.

require size $t$ parent vectors, so the parents vectors require $O(t^2)$ space. The storage for parent vectors together with the space for access histories yields $O(t^2 + Vt)$ total space.

Mellor-Crummey's offset-span labeling approach [28] reduces the size of access histories by keeping ids only for "lowest leftmost" and "lowest rightmost" readers. In this way, all dag races can be found, because a write that races with any read also races with one of the reads in the access history. The space for the access histories is therefore reduced to $O(V)$. To determine concurrency, each thread is assigned a label which consists of a sequence of offset-span pairs. The $ith$ child thread is labeled by appending the pair $[i, s]$ to its parent's label, where $s$ is the total number of children being created, the ***span***, and $i$ is called the ***offset***. The mechanism for thread joining is complicated, but the idea is that one of the pairs $[o, s]$ is replaced with $[o + s, s]$. We can check if thread $e_1$ precedes $e_2$ by checking if the threads' labels contain pairs $[o_1, s]$ and $[o_2, s]$, respectively, such that $o_1 \bmod s = o_2 \bmod s$.

The maximum size of labels is once again $O(p)$, so the space for the labels is bounded by $O(Vp)$ (assuming garbage collection). This space is also bounded by $(bp)$, so the total space of the algorithm is $O(V + \min(bp, Vp))$. Assuming that mod is a constant time operation, the time to check each memory access is just $O(p)$, the time to compare two labels.

The SP-BAGS algorithm [14], as we have seen, uses a variation on Tarjan's least common ancestor algorithm to find the logical relationships between threads. This algorithm runs in $\alpha(V, V)$ amortized time per memory access, and its disjoint set structure requires $O(V)$ space when reference counting garbage collection is used. One key idea of SP-BAGS is that by running the program in a known serial order, the size of the access history can be reduced, because the relation $\|$ is pseudotransitive (Lemma 2). SP-BAGS thus keeps only one reader per access history and so requires only $O(V)$ total space.

So far all of the algorithms we have discussed do not properly handle programs with locks. That is, they report as dag races parallel updates, even if those updates hold a lock in common. Dinning and Schonberg give a way to extend their previous work to correctly identify dag races in programs with locks [11]. The idea is to keep, for every thread id in an access history, the set of locks that were held at the time of that access. Accesses that use distinct locksets must all be recorded in the access history. Dinning and Schonberg's Lock Covers algorithm maintains access histories of size $O(tkL)$, where $k$ is the maximum number of locks held simultaneously, and $L$ is the maximum of the number of distinct lock sets used to access any particular location.

Dinning and Schonberg do not specify how this algorithm should determine concurrency. If we assume they use their earlier task recycling algorithm, then concurrency can be determined in $O(1)$ time and $O(t^2)$ space. In order to detect dag races, the algorithm must also intersect sets of locks; such an intersection requires $O(k)$ time (assuming the sets are sorted in some way). The algorithm therefore requires $O(tk^2L)$ time per memory access and $O(t^2 + tkLV)$ total space.

The All-Sets algorithm, then, can be seen as a variation of Lock Covers that achieves better asymptotic performance by using the same ideas as the original Nondeterminator—the disjoint set structure and the pseudotransitivity of $\|$. As we have seen, the algorithm uses $O(kLV)$ space and $O(L(k + \alpha(V, V)))$ amortized time per memory access.

Savage et al. [39] originally proposed the idea of using a locking discipline for race-detection purposes. Their discipline requires that every access to a variable that is shared be protected a single lock. Their model does not allow for nested parallelism or barriers. Rather, they simply assume that all accesses are in parallel with each other.[3] At each access, the set of locks that is allowed to protect the location being accessed is intersected with the currently held set of locks. This operation takes $O(k)$ time and requires the access history to hold a lockset of size $O(k)$. So Eraser takes $O(k)$ time per memory access, and requires a total of $(kV)$ space.

The Brelly algorithm can therefore be seen as an application of the idea of

[3] Actually, Eraser allows for an initial serial "initialization phase" in which a variable may be written without being protected by a lock. This phase is assumed to end as soon as an access in a different thread occurs. This access itself may constitute a race, but Eraser does not report this possibility.

locking disciplines to a general series-parallel environment. Its $O(k\,\alpha(V,V))$ amortized time per memory access and $O(kV)$ space usage are almost equivalent to Eraser's asymptotic bounds.

Others have proposed detecting races by "piggybacking" on the machine's cache coherence protocol [30, 37]. In principle, such piggybacking is only useful in detecting data races that *actually* occur in an execution. That is, the cache coherence protocol can detect when threads that actually run in parallel access the same location. To detect races based on logical relationships, these approaches must do extra work similar to the other algorithms we have seen.

Comparing times per memory access is slightly unfair, because SP-BAGS, ALL-SETS, and BRELLY all run in series, whereas the other algorithms run in parallel. The other algorithms, however, need to add extra locking in order to synchronize between updates to the access histories. This synchronization adds extra work to the program and may reduce its parallelism as well. Additionally, running the debugger in parallel means that if the input program is nondeterministic, then the debugger itself will be nondeterministic. This behavior is probably not desirable when debugging, as programmers may need to run the debugger several times if they plan on fixing race bugs one-by-one.

# Part II

# Theory of Nondeterminism

# Chapter 5

# Nondeterminism

In this chapter, we give a model for defining nondeterminism and use that model to define a hierarchy of forms of nondeterminism. The model allows programmers to define the specific form of nondeterminism that they care about for any particular program. The model is used in Chapters 7 and 8 to precisely explain the guarantees of determinacy that the Nondeterminator-2 provides.

## A model for Cilk execution

In order to describe nondeterministic program executions, we first give a formal multithreaded machine model that describes the actual execution of a Cilk program. In particular, we explain how a program execution can be viewed as a sequence of "instruction instantiations."

We can view the abstract execution machine for a multithreaded language as a (sequentially consistent [26]) shared memory together with a collection of ***interpreters***. (See [4, 9, 20] for examples of multithreaded implementations similar to this model.) Each interpreter contains ***private state*** which only it can modify. Part of its private state is a ***program counter***, which points to an instruction within the code for the program. (We assume that the code is read-only, and so where it resides is immaterial.) The ***state*** of the multithreaded machine can be viewed as a ***private state vector***, consisting of the private interpreter states, together with a ***shared***

***state vector***, consisting of the shared memory. Both state vectors may grow and shrink during execution, since new interpreters are created and destroyed, and shared memory can be allocated and freed.

Although a multithreaded execution may proceed in parallel, we consider a serialization of the execution in which only one interpreter executes at a time, but the instructions of the different interpreters may be interleaved.[1] The initial state of the machine consists of a single interpreter whose program counter points to the first instruction of the program. At each step, a nondeterministic choice among the current nonblocked interpreters is made, and the instruction pointed to by its program counter is executed.

When an instruction is executed by an interpreter, it maps the current state of the multithreaded machine to a new state.[2] There are eight types of instructions[3]:

**ALU:** Modifies only the state of the interpreter that executes it.

**READ:** Loads a value from shared memory into the local interpreter state.

**WRITE:** Stores a value into shared memory from the local interpreter state.

**LOCK:** Acquires a specified lock (special location in shared memory). Cannot be executed unless no other interpreter holds the lock.

**UNLOCK:** Releases a specified lock.

**SPAWN:** Creates a new interpreter with a specified program counter and local state. The new interpreter is a ***child*** of the original interpreter.

**SYNC:** No-op. Cannot be executed unless the interpreter has no children.

**RETURN:** Syncs, then destroys the interpreter.

[1] The fact that any parallel execution can be simulated in this fashion is a consequence of our choice of sequential consistency as the memory model. The model also assumes that single instructions are guaranteed to be atomic by the hardware, which is the case in most modern machine architectures.

[2] An instruction can formally be said to be a state to state mapping. This definition means that an instruction itself is always deterministic; we do not discuss random number generators or other forms of "serial nondeterminism."

[3] Two additional instructions, MALLOC and FREE, are discussed in Chapter 9.

In addition to performing one of these actions, executing an instruction typically causes an interpreter to modify its program counter to point to the next instruction in the program. Only an ALU instruction is allowed to modify the program counter to become anything other than the next instruction specified by the program.[4] An interpreter whose next instruction cannot be executed is said to be ***blocked***. If all interpreters are blocked, the machine is ***deadlocked***, and the execution is said to be a ***deadlock execution***.

Additionally, during the execution of a program, we can assign a unique ***interpreter name*** to each interpreter, in the following manner. The first interpreter is named by some fixed string, say "Interpreter." At each spawn, an interpreter names the newly created child interpreter by appending the number of children it has spawned to its own name. For example, the interpreter that is the third child spawned from the fourth child of the initial interpreter is named "Interpreter43."

When an instruction executes in a run of a program, it has a dynamic effect on the state of the machine. To formalize the effect of an instruction execution, we define an ***instantiation*** of an instruction to be a 3-tuple consisting of an instruction $I$, the shared memory location $l$ on which $I$ operates (if any), and the name of the interpreter that executes $I$. (Technically, this 3-tuple should probably be called a partial instantiation, as it does not specify all the values involved in the execution of $I$, but we refer to it as an instantiation for convenience.) By examining the eight types of machine instructions, we can see that when an interpreter executes an instruction, the instantiation of that instruction is entirely determined by the private state of the interpreter.

We therefore think of an ***execution*** of a program to be the sequence of instantiations resulting from running the machine model on the program. This view of executions is precisely the reason we have defined the concept of an instantiation: to make it explicit which memory locations are touched by the instructions of an execution. This formulation makes it easier to define nondeterminism.

[4] In other words, the program may not branch on a value in shared memory. It must first read that value into private memory, and then issue a branching ALU instruction.

## A model of nondeterminism

This section provides a framework for defining forms of nondeterminism, and defines a few common nondeterminacy classes. In particular, we give a formal definition of what it means for a program to have a data race.

From the English definition of the word, a program might be called "nondeterministic" if it produces differing behaviors on different executions. Many forms of nondeterminism are possible, however. Nondeterminism may be intended by the program, or it may be an accidental artifact of parallel execution. A program might behave nondeterministically "in the middle" of execution but produce a deterministic answer.

Rather than using the term "nondeterministic" ambiguously, it is desirable to distinguish between its many forms. Emrath and Padua [13] call a program ***determinate*** if it "always leads to the same results," or ***nondeterminate*** otherwise. They further divide these categories into subcategories. They call a program ***internally determinate*** if the sequence of instructions each thread executes, along with the values of the variables used by each instruction, is determinate. If a program's output is determinate, but the program is not internally determinate, Emrath and Padua say it is ***externally determinate***. A nondeterminate program is called ***associatively nondeterminate*** if the nondeterminate output is due only to lack of associativity of floating-point operations, or ***completely nondeterminate*** otherwise.

Netzer and Miller [35] use a formal model of program behavior based on Lamport's theory of concurrent systems [27] to define nondeterminism. They are specifically concerned with defining race conditions. They define a ***general race*** to occur in a program when two conflicting memory accesses are not forced to occur in a fixed order. The idea is that a general race is a bug in a program that is intended to be deterministic. A ***data race***, on the other hand, is a bug in a program that's intended to be nondeterministic, and represents only nonatomic execution of critical sections.

Netzer and Miller further distinguish both general and data races as being either "feasible" or "apparent." A ***feasible*** race is one which could occur in an actual execution of the program. An ***apparent*** race is a race that appears when only the explicit synchronization of the program is considered. Netzer and Miller say that apparent races are approximations to feasible races, and that most race detection algorithms implicitly detect apparent races.

We present our own formal model for defining types of nondeterminism. Our goal is twofold. First, we would like to be able to define a framework in which any form of nondeterminism can be defined. Rather than defining the particular forms that we think are important, our formalization makes it possible to define an unlimited number of types of nondeterminism.

Secondly, our formalism allows us to explain precisely what our proposed race detection algorithms do. We discuss program executions at the instruction level, so that the model is easy to understand. An instruction has a precisely defined meaning, and so may be easier to reason about than a model based on "events."

We observe that it does not really make sense to speak of a single execution as being nondeterministic, because nondeterminism implies that multiple executions produce varying results. Therefore, we define a *set of executions* as being deterministic or nondeterministic. Initially, the set of executions we consider are the executions that the program can generate according to the machine model. Later in this thesis, we consider other sets of executions as well.

To define a form of nondeterminism, we define an equivalence relation $\sim$ on executions. Thus, a set of executions $\mathcal{X}$ is ***nondeterministic under*** $\boldsymbol{\sim}$ if there exists executions $X_1, X_2 \in \mathcal{X}$ such that $X_1 \not\sim X_2$. Similarly, $\mathcal{X}$ is ***deterministic under*** $\boldsymbol{\sim}$ if $X_1 \sim X_2$ for all $X_1, X_2 \in \mathcal{X}$.

Using this approach, we can define many forms of nondeterminism. We discuss several common possibilities here. Rather than explicitly saying "deterministic under equivalence relation $\sim$," we often call such programs "$\sim$ deterministic."

As Emrath and Padua point out, a program may be deterministic on one input but nondeterministic on another. Since we have chosen to define forms of determinacy on sets of executions, we are implicitly discussing the determinacy of a program for a given input.

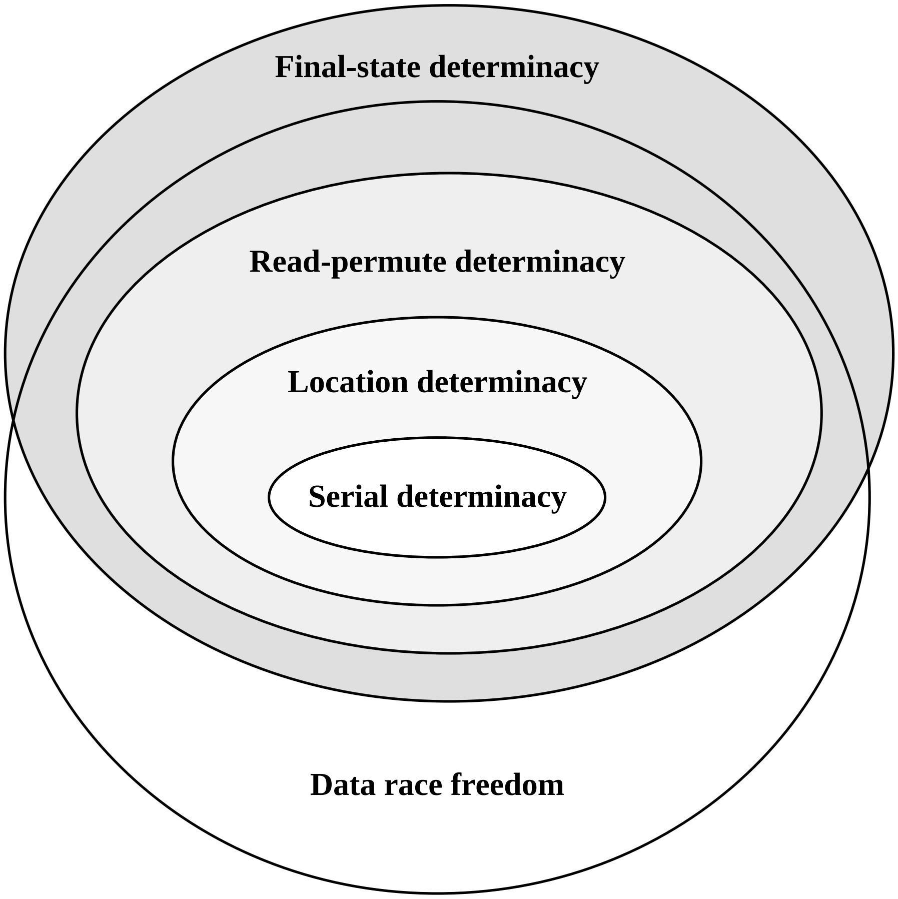


**Figure 5-1**: The hierarchy of determinacy classes. Each oval in the diagram represents the set of programs that satisfy a particular definition of determinacy.

## A hierarchy of determinism

Figure 5-1 shows the hierarchy of determinism (or nondeterminism) that we define. This chapter does not formally show the relationships between the different types of nondeterminism, but each relationship can either be inferred directly from the definition or is shown later in this thesis.

An execution is ***serial equivalent*** only to itself. Therefore, a program is ***serial deterministic*** if it generates only one execution, namely, if it is a serial program.

Recall that an execution $X$ is defined to be a sequence of instantiations, where an instantiation $x$ is a triple $\langle\ I,\ l,\ \eta\ \rangle$ consisting of instruction $I$, memory location $l$, and interpreter $\eta$. For such an instantiation, we define the selectors $\mathcal{I}$, $\mathcal{L}$, and $\mathcal{N}$ such that $\mathcal{I}(x) = I$, $\mathcal{L}(x) = l$, and $\mathcal{N}(x) = \eta$. Let us define the ***location subsequence***

$X|_l$ of location $l$ on execution $X$ to be the subsequence formed by taking all $x_i \in X$ such that $\mathcal{L}(x_i) = l$.[5] We also will use $\pi$ to denote a permutation on a set of integers.

Two executions $X = x_1 x_2 \ldots x_m$ and $Y = y_1 y_2 \ldots y_n$ are ***location equivalent*** if the following two conditions hold:

1. There exists a permutation $\pi$ such that $x_i = y_{\pi(i)}$ for all $i \in 1, 2, \ldots, n$ (and hence $n = m$).

2. For all memory locations $l$, we have $X|_l = Y|_l$.

In other words, a ***location deterministic*** program is allowed to have operations on different memory locations interleaved, but the operations on each individual memory location must be serialized in a fixed order.

We can weaken this definition of determinacy by allowing reads to be permuted.

Two executions $X = x_1 x_2 \ldots x_m$ and $Y = y_1 y_2 \ldots y_n$ are ***read permute equivalent*** if both of the following conditions are true:

1. There exists a permutation $\pi$ such that $x_i = y_{\pi(i)}$ for all $i \in 1, 2, \ldots, n$ (and hence $n = m$).

2. For all memory locations $l$, there exists a permutation $\pi_l$ such that the following two conditions hold:

   (a) $x_i \in X|_l$ if and only if $y_{\pi_l(i)} \in Y|_l$.

   (b) If $\mathcal{I}(x_i)$ or $\mathcal{I}(x_j)$ is not a READ instruction for any $x_i, x_j \in X|_l$, then $i < j \Rightarrow \pi_l(i) < \pi_l(j)$.

A ***read permute deterministic*** program, therefore, is allowed to have reads of the same memory location permuted around each other, but not around writes to that location. Read-permute determinism is what is typically meant by just the word "deterministic."

Two executions are ***final state equivalent*** if both leave the machine in the same

[5] Given a sequence $X = x_1 x_2 \ldots x_m$, another sequence $Z = z_1 z_2 \ldots z_k$ is a ***subsequence*** of $X$ if there exists a strictly increasing sequence $i_1 i_2 \ldots i_k$ of indices of $X$ such that for all $j = 1, 2, \ldots k$, we have $x_{i_j} = z_j$.

exact state after completion. Programs that are ***final state deterministic*** are also called ***determinate***.

Many more forms of determinacy exist that one might like to define. It might be useful to have a concept of "observable determinacy," meaning that only the externally observable state of the machine is determinate. Another possible form of determinacy is to allow writes to be permuted when they are part of commutative critical sections.[6] This particular form of determinacy resurfaces in Chapter 8. For now, we use our framework to define race conditions formally. Race conditions are of particular interest because they can be viewed as a "local" form of nondeterminism. Such local properties are usually easier to detect than large properties of the entire program.

A ***data race*** exists between two executions $X = x_1x_2\ldots x_m$ and $Y = y_1y_2\ldots y_n$ if there exists an integer $i$ in the range $1 \le i < \min(m, n)$ such that the following four conditions hold:

1. $x_1x_2\ldots x_{i-1} = y_1y_2\ldots y_{i-1}$,
2. $\mathcal{L}(x_i) = \mathcal{L}(x_{i+1})$,
3. $x_i = y_{i+1}$ and $x_{i+1} = y_i$,
4. $\mathcal{I}(x_i)$ or $\mathcal{I}(x_{i+1})$ is a WRITE instruction.

A program (with input) has a data race if any two of its executions have a data race between them. In other words, the program exhibits a data race when it can run a fixed sequence of instructions up to the point of the race, and then execute in either order two conflicting instructions. This definition captures the idea of "simultaneous" conflicting instructions, in light of the fact that the instructions themselves are atomic.

[6] Determinacy that allows permutation of commuting critical sections is not the same as final-state determinacy, for programs with noncommuting critical sections may still be final-state deterministic.

# Chapter 6

# Complexity of Race Detection

Ideally, we would have an algorithm to detect nondeterminacy for each form of nondeterminacy defined in Chapter 5, and programmers would use whatever algorithm best suited their own programs. In most cases, however, precise detection of nondeterminacy is extremely difficult, if not impossible. Precise detection of data races, like all nontrivial properties of programs, is undecidable. Furthermore, in this chapter we show that even in simplified models, detecting data races is computationally intractable. We argue that the Nondeterminator-2's detection of dag races is a computationally practical approximation to data-race detection.

**Theorem 10** *Detection of data races in Cilk programs is undecidable.*

*Proof:* The proof is similar to the standard programming proof of the undecidability of the halting problem. Assume there exists a serial decider `has_data_race` that takes as input a `Program P` (represented as a string). `has_data_race` returns `TRUE` if `P` has a data race, or `FALSE` if not.

Consider the program in Figure 6-1. The routine `Run_code_with_a_race()`, if executed, may exhibit a data race. If we pass the `DoOpposite` program as an argument to itself, we obtain a contradiction. For, if `has_data_race(DoOpposite)` returns `TRUE`, then `DoOpposite` returns without ever having a data race. If `has_data_race(DoOpposite)` returns `FALSE`, then `DoOpposite` executes `Run_code_with_`

```
cilk int DoOpposite(Program P)
{
  if (has_data_race(P))
    {
      return 0;
    }
  else
    {
      spawn Run_code_with_a_race();
      sync;
      return 0;
    }
}
```

**Figure 6-1**: A program used to contradict the existence of a decider for race detection.

a_race(), and so has a data race. Therefore, the serial decider has_data_race cannot exist. ■

The implication of Theorem 10 is that we cannot detect data races exactly at compile time. (We typically do not want to take the risk that our compiler may run forever.) The question, then, is whether data races can be detected exactly at run-time. Running the program does not suddenly turn an undecidable problem into a decidable one. Rather, the program itself may still run forever. If we assume that the program halts, however, then we may be able to guarantee that a detection tool would halt.

The first observation about this approach is that when the program runs, there may be portions of the code that do not execute at all due to the particular scheduling. For that code, we are back to the original problem. We can't statically find races, so we need to run that code as well and assume it terminates. Thus, if we assume that every scheduling of the program terminates, we may be able to exactly detect data races by running all possible schedulings. This approach requires $O(T!)$ time, where $T$ is the ordinary execution time of the program.

This bound, while finite, is far too expensive for a practical debugging tool. The next question, then, is whether we can ignore code that is never executed and just attempt to detect all data races in the code that gets run at least once. (This idea itself is not very well defined, but this particular discussion is intended to be informal.)

When critical sections execute, they occur in a particular order, but exactly detecting data races requires determining whether critical sections "synchronize." Consider the program in Figure 6-2(a). Whether the writes to `x` in `Write1` and `Write2` constitute a data race depends on the behavior of the `Unknown1` and `Unknown2` routines. If `Unknown1` and `Unknown2` do not affect each other's control flow, as is the case in Figure 6-2(b), then the program has a data race, and the final value of `x` may be either 1 or 2. The code in Figure 6-2(c), however, is also possible. In that code, `Unknown2` does not complete until `Unknown1` runs first. In that case, there is no data race, for the assignment `x = 2` must always occur after the assignment `x = 1`.

In general, `Unknown1` and `Unknown2` could be arbitrary operations. In order to detect whether the program in Figure 6-2 has a data race, we must determine whether `Unknown1` and `Unknown2` synchronize each other in some way. This determination can be shown to be undecidable in a similar fashion to the earlier undecidability proof.

One possible simplification is to assume that critical sections always form some kind of synchronization operation. We can model this simplification by assuming that every critical section is either an increment or a decrement of some "semaphore" variable. A ***semaphore*** variable may be incremented or decremented, but may never become negative. That is, if the semaphore is 0, then a decrement operation must wait until the semaphore becomes positive before proceeding. We will refer to this requirement as the ***semaphore constraint***.

In this model, we do not need to discern the behavior of critical sections, as they are assumed to be semaphore operations, and so we avoid that undecidable problem. We still need to discern which instruction orderings are allowed by the semaphore constraint, however. That is, in Figure 6-2(c), the statement `x = 2` must always occur after `x = 1`. If, however, there were a third parallel procedure that also incremented `done`, then `x = 2` could happen before `x = 1`, and there would be a data race.

For a program that runs in time $T$, discovering which reorderings of the instructions conform to the semaphore constraint can be reduced from a size $T$ graph problem that is NP-hard [25]. Thus, even in the simplified case where all critical sections are

```
int x;
int done;
Cilk_lockvar A;

cilk int main()
{
  done = 0;
  spawn Write1();
  spawn Write2();
  sync;
  printf("%d", x);
  return 0;
}

cilk void Write1()
{
  x = 1;
  Cilk_lock(A);
  Unknown1();
  Cilk_unlock(A);
}

cilk void Write2()
{
  Cilk_lock(A);
  Unknown2();
  Cilk_unlock(A);
  x = 2;
}
```

(a)

```
void Unknown1()
{
  done++;
}

void Unknown2()
{
  done++;
}
```

(b)

```
void Unknown1()
{
  done++;
}

void Unknown2()
{
  while (!done)
    {
      /* allow Unknown1()
         to acquire A */
      Cilk_unlock(A);
      Cilk_lock(A);
     }
}
```

(c)

**Figure 6-2**: The program in (a) may exhibit a data race on `x` depending on the behavior of `Unknown1` and `Unknown2`. (b) shows an example of these routines that leads to a race on `x`, whereas (c) shows an example that does not.

known to be semaphore operations, exact detection of data races is still computationally infeasible.

The Nondeterminator-2, therefore, does essentially the opposite: It assumes that critical sections do *not* synchronize each other in any way. In other words, the Nondeterminator-2 assumes that locks are being used only to provide atomicity, and not to implement synchronization. Thus, for the program in Figure 6-2(c), the Nondeterminator-2 reports a data race when there is none. This chapter has shown, however, that any computationally practical algorithm cannot be 100 percent accurate in its race-detection reporting. The precise meaning of the Nondeterminator's race reports is discussed and formalized in the next few chapters.

An alternate assumption also allows computationally feasible race detection algorithms. This approach only considers the particular semaphore ordering that is exhibited in one execution of the program, rather than attempting to discern other orderings. The advantage of this approach is that it only detects true data races. The problem, however, is that many data races will be missed when, as is commonly the case, critical sections do not synchronize.

# Chapter 7

# The Dag Execution Model

We have seen that detection of data races is computationally infeasible, but we have also seen that the Nondeterminator can efficiently detect dag races. In this chapter, we explain precisely why dag races are not the same thing as data races. Since the Nondeterminator-2 detects dag races, this chapter details exactly when the Nondeterminator-2 reports bugs that are not data races, and when the tool fails to report data races.

When a Cilk program executes, it generates an associated computation dag.[1] The idea is that a dag generated by a single execution contains information about other possible executions of the program. By examining the dag, we can glean information about executions other than the one that was actually run. In other words, the dag is an attempt to abstract away from a particular scheduling of threads to processors. Hence, the dag contains "logical" relationships rather than "actual" ones. These logical relationships, however, only represent the synchronization of the program due to parallel control constructs, and not any synchronization that may occur due to the operation of critical sections on shared memory.

[1] Formally, a computation dag can be constructed from an execution as follows. An initial node is created that can be considered to correspond to the initialization of the first interpreter. Whenever an interpreter executes an instruction $I$ other than a RETURN, with instantiation $x$, the interpreter creates a new vertex $x$ and adds to the dag an edge $y \to x$ from its last executed instantiation $y$ to $x$. If the instruction is a SPAWN, an additional instantiation $z$ is created (representing the initialization of the child interpreter), and the edge $y \to z$ is added to the dag. If the instruction is a RETURN, no new vertex is created, but an edge goes from $y$ to the vertex created by the next SYNC of the interpreter's parent.

A ***scheduling*** $X$ of a dag $G$ is a topological sort of the dag.[2] A scheduling is ***legal*** if, for any two LOCK statements that acquire the same lock, there is an UNLOCK of that lock in between them. A dag $G'$ is said to be a ***prefix*** of a dag $G$, if, for any nodes $x$ and $y$ such that $x \prec_G y$ and $y \in G'$, we have $x \prec_{G'} y$. A ***partial scheduling*** of $G$ is a legal scheduling of a prefix of $G$, and if any partial scheduling of $G$ can be extended to a scheduling of $G$, we say that $G$ is ***deadlock free***. Otherwise, $G$ has at least one ***deadlock scheduling***, which is a partial scheduling that cannot be extended.

A legal scheduling of a dag, therefore, is an approximation to an execution of the program. When a legal scheduling of the dag corresponds to an actual execution of the program as defined by the machine model, we say that the scheduling is a ***feasible scheduling***; otherwise, it is an ***infeasible scheduling***.

It may in fact be the case that a legal scheduling of a dag is not feasible, for two possible reasons. The first reason is demonstrated by the program and corresponding dag in Figure 7-1.[3] In particular, that dag is generated when `bar1` obtains lock `A` before `bar2`. Every scheduling of this dag contains the instantiation $x_6$ even though it does not occur in every execution. (If `bar2` obtains lock `A` before `bar1`, then the `y = 3` statement is never executed.) So $x_0 x_1 x_8 x_9 x_{10} x_{11} x_3 x_4 x_5 x_6 x_7 x_2$, for example, is a legal scheduling that is not feasible.

We call this situation the ***forced program counter anomaly***. A scheduling of a dag specifies an entire sequence of instantiations. When the machine model executes an instantiation, the model also specifies which instruction should next be executed by that interpreter. The next instruction executed by that interpreter in the dag scheduling, on the other hand, is "forced" to be the next one specified in the dag, and so may not match the one chosen by the machine model.

The other reason that legal schedulings may not be feasible is the ***forced memory***

[2] A topological sort $X$ of $G$ is a permutation of the nodes of $G$ that satisfies the constraints of the dag; if $x \prec y$ in $G$, then $x$ must occur before $y$ in $X$.

[3] Recall that the nodes of a dag are actually instantiations, not instructions. Since each instruction is executed only once in this example, we simplify notation by labeling the instructions of the program in Figure 7-1 with the instantiations $x_i$ they generate. This labeling is a further simplification because some lines of the program actually each correspond to *multiple* machine instantiations.

```
          int x;
          int y;
          Cilk_lockvar A;

          cilk int main()
          {
x0 :        x = 0;
x1 :        Cilk_lock_init(A);
            spawn foo1();
            spawn foo2();
            sync;
x2 :        printf("%d", y);
            return 0;
          }
```

```
          cilk void foo1()
          {
x3 :        Cilk_lock(A);
x4 :        x++;
x5 :        if (x == 1)
              y = 3;
x7 :        Cilk_unlock(A);
          }

          cilk void foo2()
          {
x8 :        Cilk_lock(A);
x9 :        x++;
x10 :       Cilk_unlock(A);
x11 :       y = 4;
          }
```

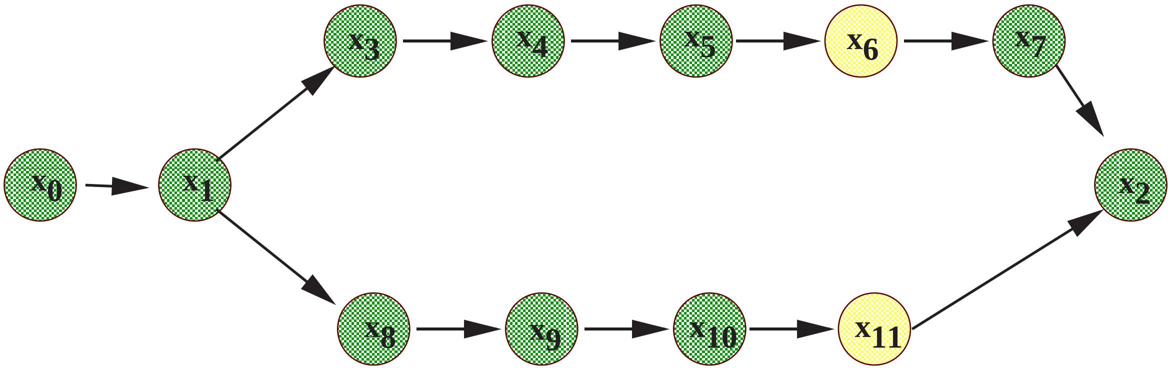


**Figure 7-1**: A program that generates a dag that exhibits the forced program counter anomaly. The dag shown here is generated when `foo1` acquires lock `A` before `foo2`. There is a dag race between the highlighted instantiations, but the program has no data race.

***location anomaly***. An instantiation contains a shared memory location. In a dag scheduling, the sequence of shared memory locations that are read is fixed. This sequence may not match the memory locations that would be read, however, if the machine model were to execute the same sequence of instructions. For example, an instruction might be "read into register 1 the contents of memory that is at the address contained in register 2." In a dag scheduling, the memory location read by this instruction's instantiation is fixed, and may not correspond to the location specified by register 2.

Figure 7-2 shows an example of a program that exhibits the forced memory loca-

```
int x[2];
int *y;
int z;
Cilk_lockvar A;

cilk int main()
{
  x[0] = 0;
  x[1] = 1;
  y = x;
  Cilk_lock_init(A);
  spawn bar1();
  spawn bar2();
  sync;
  printf("%d", x);
  return 0;
}
```

```
cilk void bar1()
{
  Cilk_lock(A);
  z = *y;
  Cilk_unlock(A);
}

cilk void bar2()
{
  Cilk_lock(A);
  (*y)++;
  Cilk_unlock(A);
}
```

**Figure 7-2**: A program that exhibits the forced memory location anomaly.

tion anomaly. The memory location that is read in the statement `z = *y` depends on whether `bar1` or `bar2` obtains lock `A` first. Any dag for this program, however, has a fixed memory location in the instantiation for the read of `*y`.

Although dag schedulings do not always correspond to machine executions, we can still consider them as executions of a ***dag execution machine***. The dag execution machine behaves similarly to the ordinary Cilk execution machine, but the program counter of each interpreter is always set to point to the next instruction in the dag, and the memory locations read are those specified in the instantiations, rather than by the instructions. When viewed as a set of dag execution machine executions, the legal schedulings of a dag form either a deterministic or nondeterministic set, according to the definitions in Chapter 5. In particular, a dag has a data race if two of its legal schedulings have a data race between them.

By definition, a dag race exists on a computation dag if two logically parallel threads access the same memory locations while holding no locks in common, and at least one of the threads writes the location. This definition of a dag race is equivalent to the definition of a data race on the set of legal schedulings of a dag. For, if two parallel threads hold no locks in common, then we can always construct a legal

```
      int max;
      int x;
      Cilk_lockvar A;

      cilk int main()
      {
x0 :    max = 0;
x1 :    Cilk_lock_init(A);
        spawn GetMax1(7);
        spawn GetMax1(3);
        sync;
x2 :    printf("%d", x);
        return 0;
      }
```

```
       cilk void GetMax1(int y)
       {
x3 :     x = max;
x4 :     Cilk_lock(A);
x5 :     if (y > max)
x6 :       max = y;
x7 :     Cilk_unlock(A);
       }

       cilk void GetMax2(int y)
       {
x8 :     Cilk_lock(A);
x9 :     if (y > max)
x10 :      max = y;
x11 :    Cilk_unlock(A);
       }
```

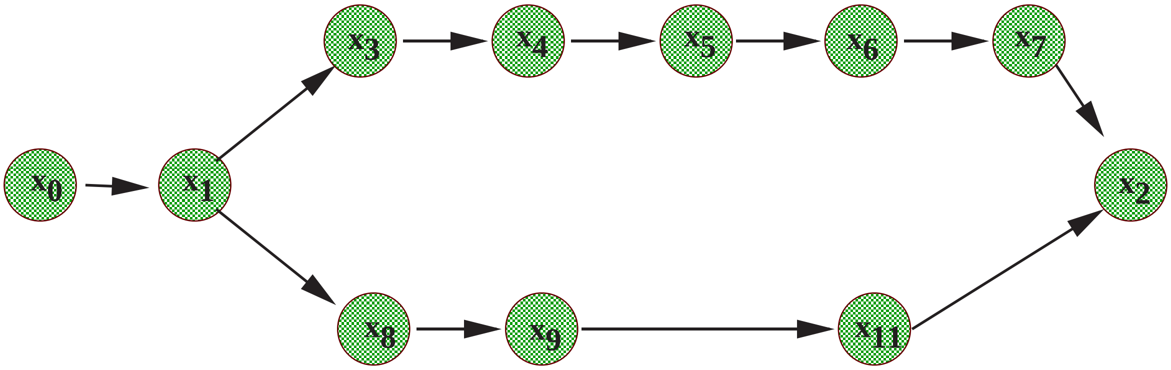


**Figure 7-3**: A program with a data race (on variable `max`) that may not appear as a dag race due to the forced program counter anomaly. The dag shown, generated when `GetMax1` acquires `A` before `GetMax2`, does not have a dag race.

scheduling of the dag by scheduling all of the predecessors of the threads, followed by the threads themselves in either order.

Since dag executions are not always machine executions, it is not surprising that dag races do not always correspond to data races in the program. Figure 7-1 shows a program that does not exhibit a data race. Indeed, the final value of `y` is always 4. The dag in Figure 7-1, however, exhibits a dag race on `y`, as the two writes to `y` are logically in parallel, and do not hold any locks in common.

Additionally, there may be data races in the program that do not appear as dag races. Such "missing races" once again may be due to the forced memory location or forced program counter anomalies. Figure 7-3 shows an example of the latter causing a data race to be missing from the dag. The program takes the maximum of two numbers in parallel, but the writes to the `max` variable depend on the order in which the critical sections are executed. The dag in Figure 7-3, for example, is generated by an execution in which `GetMax1` obtains lock `A` before `GetMax2`. In that dag, the potential write of `max` by `GetMax2` does not appear. The dag has no dag races, but there is a data race between the write of `max` in `GetMax2` and the read of `max` done in the `x = max;` statement. The final value of `x` in this program may be either 0 or 3.

There is another reason that data races may not appear as dag races that is not due to either of the aforementioned anomalies. The reason is that some code may never be executed, as discussed in Chapter 6. Figure 7-4 shows a simple example. The dag in Figure 7-4, which has no dag races, is generated by an execution where `WriteX1` obtains lock `A` before `WriteX2`. Yet clearly, if the opposite occurred, there would later be a race on the variable `y`.

In many cases, therefore, dag races are not the same as data races. Since the Nondeterminator-2 reports dag races, its reports will not exactly correspond to data races. When the computation has a dag race that is not actually a data race, the Nondeterminator-2 will report a "false positive." When the program has a data race that does not appear as a dag race in the computation, the Nondeterminator-2 will fail to report that race — a "false negative."

The Nondeterminator-2 detects dag races because, intuitively, they may sometimes be the same as data races, as the dag is an approximation to the semantics of the program. We would therefore like to answer the question: When is it that dag races actually correspond to data races?

```
        int x;
        int y;
        Cilk_lockvar A;

        cilk int main()
        {
x0 :      x = 0;
x1 :      y = 0;
x2 :      Cilk_lock_init(A);
          spawn WriteX1();
          spawn WriteX2();
          sync;
x3 :      if (x == 1)
            {
            spawn RaceY(3);
            spawn RaceY(4);
            sync;
            }
x4 :      printf("%d", y);
          return 0;
        }
```

```
        cilk void WriteX1()
        {
x5 :      Cilk_lock(A);
x6 :      x = 1;
x7 :      Cilk_unlock(A);
        }

        cilk void WriteX2()
        {
x8 :      Cilk_lock(A);
x9 :      x = 2;
x10 :     Cilk_unlock(A);
        }

        cilk void RaceY(int z)
        {
x11 :     y = z;
        }
```

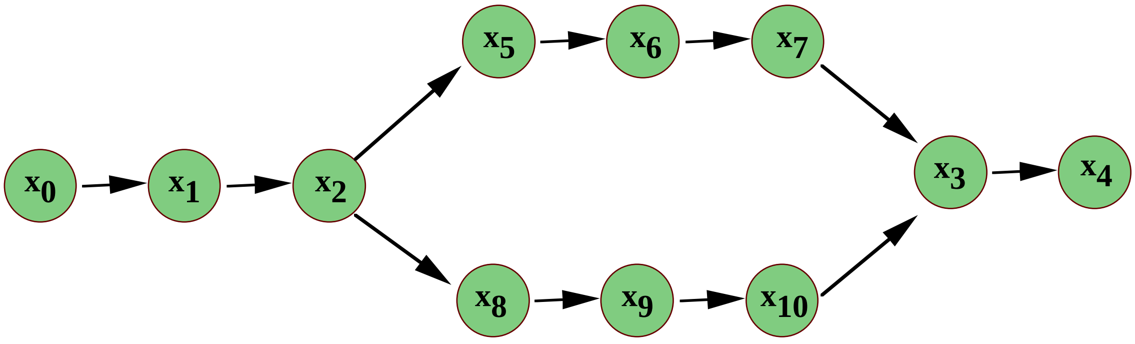


**Figure 7-4**: A program with a data race (on variable `y`) that may not appear as a dag race, because the code that exhibits the race may not be executed. Here we show the dag generated when the lock `A` is obtained first by `WriteX1` and then by `WriteX2`. As the instantiation $x_{11}$ appears nowhere in the dag, there is no dag race.

# Chapter 8

# Abelian Programs

In this chapter, we define "abelian" programs and prove that a deadlock-free abelian program has a data race if and only if every possible generated dag has a dag race.[1] Furthermore, we show that the absence of dag races in a single computation of a deadlock-free abelian program implies that the program, when run on the same input, is determinate. Thus, the Nondeterminator-2 can verify that a deadlock-free abelian program is determinate for a given input.

In practice, most programs that use locks in any significant way are not abelian. The existence of the class of abelian programs is itself interesting, however. This class shows that there is in fact a formal relationship between dag races and data races. Furthermore, the guarantee that the Nondeterminator-2 provides for abelian programs is a somewhat remarkable result, because programs that use locks are generally "inherently nondeterministic;" that is, they are read-permute nondeterministic. Nonetheless, abelian read-permute nondeterministic programs can be shown to always produce the same final machine state.

## Abelian programs

The program in Figure 8-1 is an example of an abelian program. The program is read-permute nondeterministic, as the updates to `x` may happen in different orders.

[1] Some of the results in this chapter appear in [6].

```
int x;
Cilk_lockvar A;

cilk int main()
{
  x = 0;
  Cilk_lock_init(A);
  spawn UpdateX1();
  spawn UpdateX2();
  sync;
  printf("%d", x);
  return 0;
}
```

```
cilk void UpdateX1()
{
  Cilk_lock(A);
  x += 2;
  Cilk_unlock(A);
}

cilk void UpdateX2()
{
  Cilk_lock(A);
  x += 3;
  Cilk_unlock(A);
}
```

**Figure 8-1**: An example of an abelian program. This particular program has no data races or deadlocks, and so is determinate.

In other words, x has an "intermediate" value that is nondeterministically either 2 or 3, but x always ends with a value of 5.

The critical sections in the program in Figure 8-1 obey the following strict definition of commutativity: Two critical sections $R_1$ and $R_2$ ***commute*** if, beginning with any reachable program state $S$, the execution of $R_1$ followed by $R_2$ yields the same state $S'$ as the execution of $R_2$ followed by $R_1$; and furthermore, in both execution orders, each critical section must execute the identical sequence of instructions on the identical memory locations.[2] Thus, not only must the program state remain the same, the same accesses to shared memory must occur, although the values returned by those accesses may differ.[3] The program in Figure 8-1 also exhibits "properly nested locking." Locks are ***properly nested*** if any thread that acquires a lock A and then a lock B releases B before releasing A. We say that a program is ***abelian*** if any pair of parallel critical sections that are protected by the same lock commute, and all

[2] It may be the case that even though $R_1$ and $R_2$ are in parallel, they cannot appear adjacent in any execution, because a lock that is acquired preceding $R_1$ and released after $R_1$ is also acquired by $R_2$ (or vice versa). Therefore, we require the additional technical condition that the execution of $R_1$ followed by any prefix $R_2'$ of $R_2$ generates for $R_2'$ the same instructions operating on the same locations as executing $R_2'$ alone. This requirement is used in the proof of deadlock in Appendix A.

[3] By requiring that the entire machine state $S$ remain the same, we mean that the private states of the interpreters that execute $R_1$ and $R_2$, in addition to the shared memory $M$, must be the same regardless of the execution order of the regions. This requirement implies that any temporary variables that are used to store intermediate values should be reset at the end of every critical region, in order to satisfy the commutativity definition. In practice, of course, temporary variables that are not live at the end of critical regions can be left with nondeterministic values.

locks in the program are properly nested.

The idea that critical sections should commute is natural. A programmer presumably locks two critical sections with the same lock not only because he intends them to be atomic, but because he intends them to "do the same thing" no matter in what order they are executed. The programmer's notion of commutativity is usually less restrictive, however, than what our definition allows. First, both execution orders of two critical sections may produce distinct program states that the programmer nevertheless views as equivalent. Our definition insists that the program states be identical. Second, even if they leave identical program states, the two execution orders may cause different memory locations to be accessed. Our definition demands that the same memory locations be accessed.

In practice, therefore, most programs are not abelian, but abelian programs nevertheless form a nontrivial class of nondeterministic programs that can be checked for determinacy. For example, programs that use locking to accumulate values atomically, such as a histogram program, fall into this class. Additionally, all programs that don't use locks at all are abelian. Although abelian programs form an arguably small class in practice, the algorithms that we present in this thesis can provide guarantees of determinacy for abelian programs that are not provided by any other existing race-detectors for *any* class of lock-employing programs.

The converse of the determinacy guarantee is not true. That is, a program may have a data race, but later deterministically overwrite that value, resulting in a deterministic final memory state. Also, once a dag race is found, then later parts of the dag may once again exhibit the forced memory location or forced program counter anomalies. The guarantee, therefore, is that any computation dag of a deadlock-free abelian program at least contains a dag race corresponding to the "first" data race of the program (if a data race exists at all).

## Proof of the Nondeterminator-2's determinacy guarantee

The proof of the determinacy guarantee centers around "regions" of instantiations, which are sequences of instantiations executed by a single interpreter. Precisely, a

***region*** is either a single instantiation other than a LOCK or UNLOCK instruction, or a sequence of instantiations that comprise a critical section (including the LOCK and UNLOCK instantiations themselves).[4] Every instantiation belongs to at least one region and may belong to many. Since a region is a sequence of instantiations, it is determined by a particular execution of the program and not by the program code alone. We define the ***nesting count*** of a region $R$ to be the maximum number of locks that are acquired in $R$ and held simultaneously at some point in $R$.

Since we are only concerned with the final memory states of feasible schedulings, we define two legal schedulings of $G$ to be ***equivalent*** if both are infeasible, or both are feasible and have the same final memory state. An alternate definition of commutativity, then, is that two regions $R_1$ and $R_2$ commute if, beginning with any reachable machine state $S$, the instantiation sequences $R_1 R_2$ and $R_2 R_1$ are equivalent.

The proof of the equivalence of dag race freedom and final-state determinism proceeds as follows. Starting with a dag-race free, deadlock-free computation $G$ resulting from the execution of an abelian program, we first prove that adjacent regions in a legal scheduling of $G$ can be commuted. Second, we show that regions that are spread out in a legal scheduling of $G$ can be grouped together. Third, we prove that all legal schedulings of $G$ are feasible and yield the same final memory state. Finally, we prove that all executions of the abelian program generate the same computation and hence the same final memory state.

**Lemma 11 (Reordering)** *Let $G$ be a dag-race free, deadlock-free computation resulting from the execution of an abelian program. Let $X$ be some legal scheduling of $G$. If regions $R_1$ and $R_2$ appear adjacent in $X$, i.e., $X = X_1 R_1 R_2 X_2$, and $R_1 \parallel R_2$, then the two schedulings $X_1 R_1 R_2 X_2$ and $X_1 R_2 R_1 X_2$ are equivalent.*

*Proof:* We prove the lemma by double induction on the nesting count of the regions. Our inductive hypotheses is the theorem as stated for regions $R_1$ of nesting count $i$ and regions $R_2$ of nesting count $j$.

[4] The instantiations within a critical section must be serially related in the dag, as we disallow parallel control constructs while locks are held.

Base case: $i = 0$. Then $R_1$ is a single instantiation. Since $R_1$ and $R_2$ are adjacent in $X$ and are parallel, no instantiation of $R_2$ can be guarded by a lock that guards $R_1$, because any lock held at $R_1$ is not released until after $R_2$. Therefore, since $G$ is dag-race free, either $R_1$ and $R_2$ access different memory locations or $R_1$ is a READ and $R_2$ does not write to the location read by $R_1$. In either case, the instantiations of each of $R_1$ and $R_2$ do not affect the behavior of the other, so they can be executed in either order without affecting the final memory state.

Base case: $j = 0$. Symmetric with above.

Inductive step: In general, $R_1$ has nesting count $i \geq 1$, and is of the form LOCK(A) $\cdots$ UNLOCK(A). $R_2$ of count $j \geq 1$ has the form LOCK(B) $\cdots$ UNLOCK(B). If A = B, then $R_1$ and $R_2$ commute by the definition of abelian. Otherwise, there are three possible cases.

Case 1: Lock A appears in $R_2$, and lock B appears in $R_1$. This situation cannot occur, because it implies that $G$ is not deadlock free, a contradiction. To construct a deadlock scheduling, we schedule $X_1$ followed by the instantiations of $R_1$ up to (but not including) the first LOCK(B). Then, we schedule the instantiations of $R_2$ until a deadlock is reached, which must occur, since $R_2$ contains a LOCK(A) (although the deadlock may occur before this instantiation is reached).

Case 2: Lock A does not appear in $R_2$. We start with the sequence $X_1 R_1 R_2 X_2$ and commute pieces of $R_1$ one at a time with $R_2$: first, the instantiation UNLOCK(A), then the (immediate) subregions of $R_1$, and finally the instantiation LOCK(A). The instantiations LOCK(A) and UNLOCK(A) commute with $R_2$, because A does not appear anywhere in $R_2$. Each subregion of $R_1$ commutes with $R_2$ by the inductive hypothesis, because each subregion has lower nesting count than $R_1$. After commuting all of $R_1$ past $R_2$, we have an equivalent execution $X_1 R_2 R_1 X_2$.

Case 3: Lock B does not appear in $R_1$. Symmetric to Case 2. ■

**Lemma 12 (Region grouping)** *Let $G$ be a dag-race free, deadlock-free computation resulting from the execution of an abelian program. Let $X$ be some legal scheduling of $G$. Then, there exists an equivalent scheduling $X'$ of $G$ in which the instantiations* *of every region are contiguous.*

*Proof:* We create $X'$ by grouping the regions in $X$ one at a time. Each grouping operation does not destroy the grouping of already grouped regions, so eventually all regions are grouped.

Let $R$ be a noncontiguous region in $X$ that completely overlaps no other noncontiguous regions in $X$. Since region $R$ is noncontiguous, other regions parallel with $R$ must overlap $R$ in $X$. We first remove all overlapping regions that have exactly one endpoint (an endpoint is the bounding LOCK or UNLOCK of a region) in $R$, where by "in" $R$, we mean appearing in $X$ between the endpoints of $R$. We shall show how to remove regions that have only their UNLOCK in $R$. The technique for removing regions with only their LOCK in $R$ is symmetric.

Consider the partially overlapping region $S$ with the leftmost UNLOCK in $R$. Then all subregions of $S$ that have any instantiations inside $R$ are completely inside $R$ and are therefore contiguous. We remove $S$ by moving each of its (immediate) subregions in $R$ to just left of $R$ using commuting operations. Let $S_1$ be the leftmost subregion of $S$ that is also in $R$. We can commute $S_1$ with every instruction $I$ to its left until it is just past the start of $R$. There are three cases for the type of instruction $I$. If $I$ is not a LOCK or UNLOCK, it commutes with $S_1$ by Lemma 11 because it is a region in parallel with $S_1$. If $I =$ LOCK(B) for some lock B, then $S_1$ commutes with $I$, because $S_1$ cannot contain LOCK(B) or UNLOCK(B). If $I =$ UNLOCK(B), then there must exist a matching LOCK(B) inside $R$, because $S$ is chosen to be the region with the leftmost UNLOCK without a matching LOCK. Since there is a matching LOCK in $R$, the region defined by the LOCK/UNLOCK pair must be contiguous by the choice of $R$. Therefore, we can commute $S_1$ with this whole region at once using Lemma 11.

We can continue to commute $S_1$ to the left until it is just before the start of $R$. Repeat for all other subregions of $S$, left to right. Finally, the UNLOCK at the end of $S$ can be moved to just before $R$, because no other LOCK or UNLOCK of that same lock appears in $R$ up to that UNLOCK.

Repeat this process for each region overlapping $R$ that has only an UNLOCK in $R$. Then, remove all regions that have only their LOCK in $R$ by pushing them to just after $R$ using similar techniques. Finally, when there are no more unmatched LOCK or UNLOCK instantiations in $R$, we can remove any remaining overlapping regions by pushing them in either direction to just before or just after $R$. The region $R$ is now contiguous.

Repeating for each region, we obtain an execution $X'$ equivalent to $X$ in which each region is contiguous. ■

**Lemma 13** *Let $G$ be a dag-race free, deadlock-free computation resulting from the execution of an abelian program. Then every legal scheduling of $G$ is feasible and yields the same final memory state.*

*Proof:* Let $X$ be the execution that generates $G$. Then $X$ is a feasible scheduling of $G$. We wish to show that any legal scheduling $Y$ of $G$ is feasible. We shall construct a set of equivalent schedulings of $G$ that contain the schedulings $X$ and $Y$, thus proving the lemma.

We construct this set using Lemma 12. Let $X'$ and $Y'$ be the schedulings of $G$ with contiguous regions that are obtained by applying Lemma 12 to $X$ and $Y$, respectively. From $X'$ and $Y'$, we can commute whole regions using Lemma 11 to put their threads in the serial depth-first order specified by $G$, obtaining schedulings $X''$ and $Y''$. We have $X'' = Y''$, because a computation has only one serial depth-first scheduling. Thus, all schedulings $X$, $X'$, $X'' = Y''$, $Y'$, and $Y$ are equivalent. Since $X$ is a feasible scheduling, so is $Y$, and both have the same final memory state. ■

**Lemma 14** *Let $G$ be a dag-race free, deadlock-free computation resulting from the execution of an abelian program. Then every machine execution of the program (on the same input) generates the same dag $G$.*

*Proof:* Let $X$ be the original machine execution that generated $G$. Let $Y$ be an arbitrary execution of the same program. Let $H$ be the computation generated by $Y$, and let $H_i$ be the prefix of $H$ that is generated by the first $i$ instantiations of $Y$. If $H_i$ is a prefix of $G$ for all $i$, then $H = G$, proving the lemma. Otherwise, assume for contradiction that $i_0$ is the largest value of $i$ for which $H_i$ is a prefix of $G$. Suppose that the $(i_0 + 1)$st instantiation of $Y$ is executed by an interpreter with name $\eta$. We shall derive a contradiction through the creation of a new legal scheduling $Z$ of $G$. We construct $Z$ by starting with the first $i_0$ instantiations of $Y$, and next adding the successor of $H_{i_0}$ in $G$ that is executed by interpreter $\eta$. We then complete $Z$ by adding, one by one, any nonblocked instantiation from the remaining portion of $G$. One such instantiation always exists because $G$ is deadlock free. By Lemma 13, the scheduling $Z$ that results is a feasible scheduling of $G$. We thus have two feasible schedulings that are identical in the first $i_0$ instantiations but that differ in the $(i_0+1)$st instantiation. In both schedulings the $(i_0 + 1)$st instantiation is executed by interpreter $\eta$. But, the state of the machine is the same in both $Y$ and $Z$ after the first $i_0$ instantiations, which means that the $(i_0 + 1)$st instantiation must be the same for both, which is a contradiction. ■

**Theorem 15** *An abelian Cilk program that produces a deadlock-free computation with no dag races is determinate.*

*Proof:* Combine Lemma 13 and Lemma 14.

**Theorem 16** *A deadlock-free computation produced by an abelian Cilk program has a dag race if and only if the program has a data race.*

*Proof:* (⇐) If a deadlock-free computation has no dag races, then from Lemma 14, every machine execution generates the same dag, so every such execution is a scheduling of that dag. Thus, if two machine executions have a data race between them, then there is also a dag race between them, which is a contradiction.

(⇒) Let $G$ be a deadlock-free computation of an abelian program with a dag race that is generated by an execution $X$ of the program. Say that the dag race occurs between instantiations $x$ and $y$. Let $Z = Z_1xyZ_2$ be a legal scheduling of $G$ in which $x$ and $y$ occur adjacently. (Such a scheduling must exist, because $x$ and $y$ are in parallel and have no locks in common by definition of dag race, and the computation has no deadlocks.)

We attempt to use the techniques of Lemma 13 to commute $X$ into the form of $Z$. That is, we commute $X$ into the depth-first scheduling and then commute that into $Z$. We show that each step either succeeds or yields a feasible data race.

If all steps succeed, then $Z$ is feasible. Since $x$ and $y$ are logically in parallel, they execute on different interpreters. Moreover, since the instantiation of an instruction depends only on the private state of its interpreter, changing the order of execution of $\mathcal{I}(x)$ and $\mathcal{I}(y)$ does not affect the instantiations of either of those instructions. Therefore, $Z' = Z_1yx$ is a feasible partial execution of the program, and so the program has a data race between executions $Z$ and $Z'$.

The only place where the technique of Lemma 13 can fail is in the base case of the proof of Lemma 11, as that is the only portion that depends on dag-race freedom. In that case, we have a scheduling $X_1R_1R_2X_2$ that is equivalent to the original execution $X$ (and so is feasible), with $R_1$ a single instantiation, say $x'$, and $R_1 \parallel R_2$. The regions $R_1$ and $R_2$ can successfully be commuted unless $x'$ writes a location accessed by $R_2$ or $x'$ reads a location written by $R_2$. Let $y'$ be the first such conflicting instantiation in $R_2$. Then, we can commute $x'$ until it is adjacent with $y'$, yielding a feasible scheduling $X_1'x'y'X_2'$. By the same argument as above, $X_1'y'x'$ is also a feasible (partial) execution of the program, and so the program has a data race. A symmetric argument can be made when $R_2$ consists of a single instantiation by choosing the last conflicting instantiation in $R_1$. ■

All of the results so far have assumed a deadlock-free computation, but in general, a deadlock-free computation is not equivalent to a deadlock-free program. Fortunately, the following lemma shows that the programmer does not need to worry about this distinction when applying Theorems 15 and 16. The proof of this lemma is complicated and so is left to Appendix A.

**Lemma 17** *Let $G$ be a dag-race free computation generated by an abelian program. $G$ is deadlock free if and only if the program is deadlock free (on the same input).* ■

**Corollary 18** *If the* All-Sets *algorithm detects no data races in an execution of a deadlock-free abelian Cilk program, then the program running on the same input has no data races and is determinate.*

*Proof:* Combine Theorems 3 and 15 and Lemma 17. ■

**Corollary 19** *If the* Brelly *algorithm detects no violations of the umbrella discipline in an execution of a deadlock-free abelian Cilk program, then the program run on the same input has no data races and is determinate.*

*Proof:* Combine Theorems 5, 8, and 15 and Lemma 17. ■

# Part III

# Using the Nondeterminator-2

# Chapter 9

# Implementation Issues

In this chapter, we discuss practical issues surrounding the implementation and use of the Nondeterminator-2. We explain how to catch dag races involving the dynamic memory allocator, and show that memory cannot be recycled without the risk of missing races. We provide some heuristics for reducing the number of false reports that the Nondeterminator-2 may produce in the case of nonabelian programs, when dag races may not really be data races. Finally, we give some timings of the Nondeterminator-2 on a selection of Cilk codes, which show that the algorithms roughly conform to their theoretical bounds in practice. On all of our sample codes, BRELLY is fast enough to be used as an interactive debugger, but ALL-SETS sometimes runs too slow to be practical.

## Dynamic memory allocation

Cilk provides the routines `Cilk_malloc()` and `Cilk_free()` to dynamically allocate and deallocate shared memory. (We refer to these as single instructions MALLOC and FREE in the Cilk machine model.) The first observation is that these routines may in fact be involved in data races. For example, a read of `*x` occurring in parallel with the command `Cilk_free(x)` constitutes a race. If the FREE instruction happens first, then the value in `*x` might become garbage before it is read.[1]

[1] The semantics of FREE allow it to write garbage into memory, although in practice the memory really only changes after it is allocated again.

The solution is to treat the FREE instruction like a write when it occurs. In other words, when a FREE occurs, it is compared with all the past accesses in the shadow space to check for races. After the FREE occurs, any later access to the freed memory (other than a MALLOC) is an error, regardless of whether the access is a serial or parallel access. The FREE instruction therefore puts the special tag `FREE_ID` into the shadow space. If a later access observes this value, a bug is reported.

Once memory is freed, later accesses to it are always incorrect, regardless of which locks are held. Therefore, after memory is freed, the history of which locks have been used to access that memory is no longer needed. The Nondeterminator-2's internal memory, which is used to store that information, can thus be deallocated as well. This approach maintains the convenient property that the internal memory for storing lock sets is only allocated as long as the user's memory is allocated.

When memory is about to be allocated via a MALLOC statement, the shadow space contains `FREE_ID`. (The memory allocator is trusted to be correct.) MALLOC simply overwrites `FREE_ID` with the id of whatever thread it's running in. In this way, false positives are not reported when memory is reused. For example, consider two threads $e_1 \parallel e_2$:

| Thread $e_1$ | Thread $e_2$ |
|---|---|
| `*x = 5;` | `y = Cilk_malloc(...)` |
| `Cilk_free(x);` | `*y = 6;` |

Even though $e_1 \parallel e_2$, it is possible that in a particular execution, $e_1$ runs before $e_2$, and that the address returned from `Cilk_malloc()` and assigned to `y` is the same as the address contained in `x`. It would therefore appear that there is a dag race between the two writes to that address, `*x = 5` and `*y = 6`, as those two writes are logically in parallel. The writes do not actually constitute a race, however, because if the `Cilk_malloc()` statement were executed before the `Cilk_free(x)` statement, the memory allocator would assure that `Cilk_malloc()` would return an address different from `x`. The protocol for the Nondeterminator-2 we have described handles this situation correctly, because the `Cilk_malloc()` statement puts the ID for thread

$e_2$ in the shadow space, and so the `*y = 6` is a serial access and does not appear to be a race with the write to `*x`.

This approach also catches races involving the MALLOC statement itself. That is, by writing the current thread id into newly allocated memory, dag races can be caught if that memory is written in parallel.

There is, however, a problem with the approach we have described. Consider the following example, which is similar to the one above, but in which $e_2$ writes `*x` rather than `*y`.

| Thread $e_1$ | Thread $e_2$ |
|---|---|
| `*x = 5;` | `y = Cilk_malloc(...)` |
| `Cilk_free(x);` | `*x = 6;` |

In this example, there is always a race between the writes to `*x`. This race may be missed, however, if the `Cilk_malloc()` statement returns the same address as `x`. Then the `*x = 6` appears to be writing newly allocated memory, rather than writing to the data pointed to by `x`.

In order to distinguish this case from the previous one, we need some way to distinguish between writing to "`*x`" and "`*y`" even though both end up writing the same memory location. Making this distinction requires some understanding of the meaning of the program, rather than just monitoring of the memory locations accessed. This sort of alias analysis is typically very difficult, and we do not attempt to do it.

Rather, our solution is very simple. The Nondeterminator-2 does not recycle memory. That is, when the memory allocator is run in debugging mode, it assures that `Cilk_malloc()` never returns memory that has previously been allocated. When memory is freed, it is simply left in the free state forever. In this way, memory is never aliased, and the problem in the previous example cannot occur.

The justification for this approach is that in modern machines, memory (and virtual address space) is large and cheap. It is acceptable to use a lot of memory when debugging; memory is still be recycled when the application is in production mode. If users do not have enough memory, they can simply turn this feature of the

Nondeterminator-2 off, and go back to recycling memory. In that case, however, the dag race in the last example will be missed.

## Reducing false race reports

As we have seen, some dag races may not correspond to data races if they are artifacts of other races or of noncommutative critical sections. Other researchers have attempted to algorithmically identify "first races," as compared to later artifacts of those races [33, 34]. While we do not attempt anything of this magnitude, we do implement several tricks that can make the race reports of the Nondeterminator-2 more manageable for the user.

The first trick is to avoid reporting the "same" race more than once. When a race is reported, we enter all of the involved line numbers and file names into a hash table. If we later encounter a race with the same lines in the same files, we don't report it, as it is assumed to be another instance of the same race. This feature is essential for making the number of races reported be manageable; without it, a single race, executed over and over again, could produce thousands of lines of debugging reports.

The Brelly algorithm has an additional problem of reporting multiple races. If an unprotected umbrella is discovered, that umbrella may potentially be reported once for every access in the umbrella (other than the first one). Rather than reporting all of these separately, the Brelly algorithm should group all the accesses together and report them all at once. In some cases, it is possible to determine that some subset of the accesses actually constitutes a dag race, and those accesses can be reported in preference to the entire umbrella. See [5] for more details.

When false reports due to infeasible dag races occur, we would like to provide some way for the user to inform the Nondeterminator-2 that these races are infeasible, so that it can avoid reporting them in future executions. One approach is to allow the user to "turn off" the Nondeterminator-2's memory checking, so that certain memory accesses are ignored. User annotation can either be done lexically via a compiler pragma or dynamically by setting a global flag. While this approach may reduce race reports, it requires users to manually assure themselves that there are no races involving the ignored accesses.

A solution that requires less verification from the user is to use of fake locks—locks that are acquired and released only in debugging mode, as in the implicit R-LOCK fake lock. The user can then protect accesses involved in infeasible dag races using a common fake lock. Fake locks reduce the number of false reports made by the Nondeterminator-2, and they require the user to manually check for data races only between critical sections locked by the same fake lock.

A particularly common cause of false reports is "publishing." One thread allocates a heap object, initializes it, and then "publishes" it by atomically making a field in a global data structure point to the new object so that the object is now available to other threads. If a logically parallel thread now accesses the object in parallel through the global data structure, an infeasible dag race occurs between the initialization of the object and the access after it was published.

Fake locks do not seem to help much with the publishing problem, because it is hard for the initializer to know all the other threads that may later access the object, and we do not wish to suppress data races among those later accesses. One possible solution is to allow users to explicitly put in PUBLISH statements in the program, to declare that memory is being published. The effect of a PUBLISH statement is to erase the history of past accesses that is contained in the shadow space. Since parallel threads were unable to access the memory up to the point of the PUBLISH statement, accesses before that statement cannot be involved in races.

There are some practical difficulties in using PUBLISH in C. The size of structures may not be known statically, so the user may be required to supply the size. Furthermore, there is no way to specify that structures that are nested via pointers are all part of the same "object." The user must therefore explicitly issue a PUBLISH statement for each nested pointer data structure. Publishing of objects could be more elegantly handled in a strongly-typed language. A possible solution for C is to use checkpointing technology, which is able to automatically trace through entire data structures [40]. Even then, the semantics of PUBLISH could be difficult to express if only parts of a data structure are being published.

## Timings of the Nondeterminator-2

In this section, we give some experimental measurements of the performance of the Nondeterminator-2.[2] As it is a debugging tool, the Nondeterminator-2 does not need to achieve absolutely optimal performance. Rather, it just needs to be fast enough to use in an interactive debugging environment.

Our implementations of All-Sets and Brelly have not yet been optimized, and so the timings presented here are preliminary; better performance than what we report here is likely to be possible. In particular, our current implementation treats every READ like an ACCESS with the fake R-LOCK, as described in Chapter 2. This approach requires an allocation of a lock set at every read operation. We expect that the running time of both algorithms could be greatly improved if we optimized the common case of reads with no locks held.

According to Theorem 4, the factor by which All-Sets slows down a program is roughly $\Theta(Lk)$ in the worst case, where $L$ is the maximum number of distinct lock sets used by the program when accessing any particular location, and $k$ is the maximum number of locks held by a thread at one time. According to Theorem 9, the worst-case slowdown factor for Brelly is about $\Theta(k)$. In order to compare our experimental results with the theoretical bounds, we characterize our four test programs in terms of the parameters $k$ and $L$:[3]

`maxflow`: A maximum-flow code based on Goldberg's push-relabel method [17]. Each vertex in the graph contains a lock. Parallel threads perform simple operations asynchronously on graph edges and vertices. To operate on a vertex $u$, a thread acquires $u$'s lock, and to operate on an edge $(u, v)$, the thread acquires both $u$'s lock and $v$'s lock (making sure not to introduce a deadlock). Thus, for this application, the maximum number of locks held by a thread is $k = 2$, and $L$ is at most the maximum degree of any vertex.

`n-body`: An $n$-body gravity simulation using the Barnes-Hut algorithm [1]. In one phase of the program, parallel threads race to build various parts of an "octtree"

[2] Some of the results in this section appear in [6].

[3] These characterizations do not count the implicit fake R-LOCK used by the detection algorithms.

|  | Parameters |  |  | Time (sec.) |  |  | Slowdown |  |
|---|---|---|---|---|---|---|---|---|
| Program | input | $k$ | $L$ | orig. | All. | Br. | All. | Br. |
| `maxflow` | sp. 1K | 2 | 32 | 0.05 | 30 | 3 | 590 | 66 |
|  | sp. 4K | 2 | 64 | 0.2 | 484 | 14 | 2421 | 68 |
|  | d. 256 | 2 | 256 | 0.2 | 263 | 15 | 1315 | 78 |
|  | d. 512 | 2 | 512 | 2.0 | 7578 | 136 | 3789 | 68 |
| `n-body` | 1K | 1 | 1 | 0.6 | 47 | 47 | 79 | 78 |
|  | 2K | 1 | 1 | 1.6 | 122 | 119 | 76 | 74 |
| `bucket` | 100K | 1 | 1 | 0.3 | 22 | 22 | 74 | 73 |
| `rad` | iter. 1 | 2 | 65 | 1.2 | 109 | 45 | 91 | 37 |
|  | iter. 2 | 2 | 94 | 1.0 | 179 | 45 | 179 | 45 |
|  | iter. 5 | 2 | 168 | 2.8 | 773 | 94 | 276 | 33 |
|  | iter. 13 | 2 | 528 | 9.1 | 13123 | 559 | 1442 | 61 |

**Figure 9-1**: Timings of our implementations on a variety of programs and inputs, run on an UltraSPARC I. (The input parameters are given as sparse/dense and number of vertices for `maxflow`, number of bodies for `n-body`, number of elements for `bucket`, and iteration number for `rad`.) The parameter $L$ is the maximum number of distinct lock sets used while accessing any particular location, and $k$ is the maximum number of locks held simultaneously. Running times for the original optimized code, for All-Sets, and for Brelly are given, as well as the slowdowns of All-Sets and Brelly as compared to the original running time.

data structure. Each part is protected by an associated lock, and the first thread to acquire that lock builds that part of the structure. As the program never holds more than one lock at a time, we have $k = L = 1$.

`bucket`: A bucket sort [8, Section 9.4]. Parallel threads acquire the lock associated with a bucket before adding elements to it. This algorithm is analogous to the typical way a hash table is accessed in parallel. For this program, we have $k = L = 1$.

`rad`: The radiosity application (discussed further in Chapter 10). The code locks a "patch" of the scene along with the "surface" that the patch is on, so that $k = 2$, and $L$ is the maximum number of patches per surface, which increases at each iteration as the rendering is refined.

Figure 9-1 shows the preliminary results of our experiments on the test codes. These results indicate that the performance of All-Sets is indeed dependent on the parameter $L$. Essentially no performance difference exists between All-Sets and Brelly when $L = 1$, but All-Sets gets progressively worse as $L$ increases. On all of our test programs, Brelly runs fast enough to be useful as a debugging tool. In some cases, All-Sets is as fast, but in other cases, the overhead of All-Sets is too extreme (iteration 13 of `rad` takes over 3.5 hours) to allow interactive debugging.

# Chapter 10

# Parallel Radiosity

In this chapter, we describe our experiences parallelizing a large, real-world radiosity application. We view this application as a case study for the usefulness of the Nondeterminator-2. We used the Nondeterminator-2 to minimize the amount of the radiosity code that we needed to examine and understand. Figure 10-1 shows the speedup of our Cilk code as compared to the original optimized C version. With less than 5 percent of the code from the original version changed, the entire application achieves a speedup of 5.97 on 8 processors. Furthermore, the Nondeterminator-2 gives us a high degree of confidence that the code is actually data-race free.

## Goals of parallelizing radiosity

Radiosity is a graphics algorithm for modeling light in diffuse environments. It is an irregular application, and therefore the computation is difficult to balance statically across processors. That is, the area where the majority of the CPU time is spent depends on the input scene, and varies dynamically as the lighting is calculated. In order to get good performance on a parallel machine, the CPU time must be balanced evenly across all processors, so that all processors are utilized fully. This balancing is difficult to do at compile time when the behavior of the computation is difficult to predict.

Cilk provides a dynamic scheduler which balances tasks across processors using

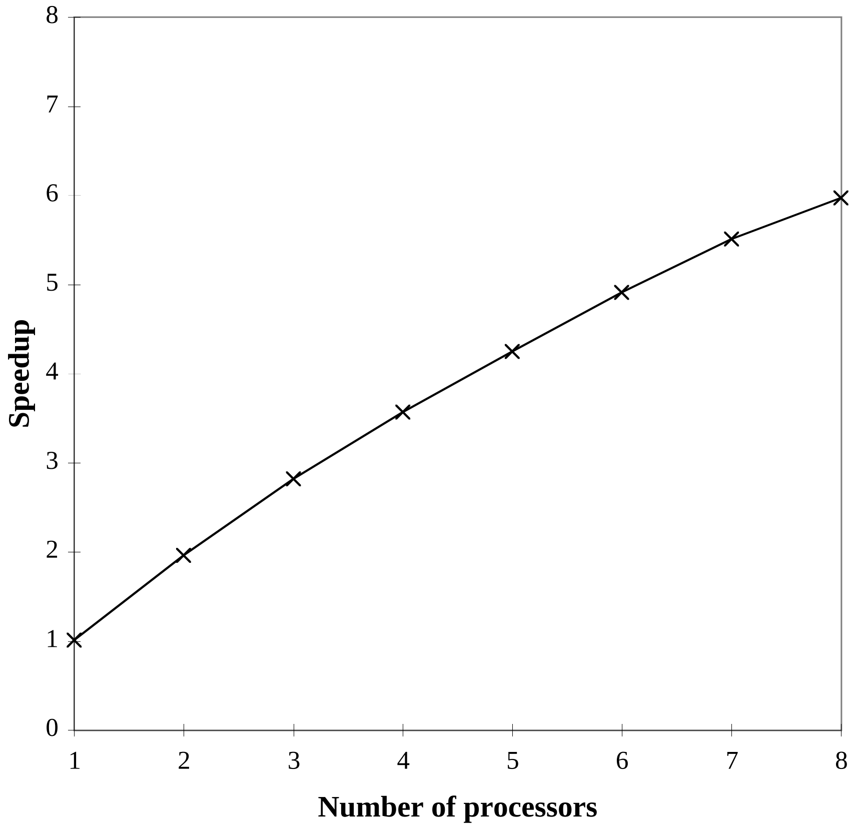


**Figure 10-1**: Speedup of the `rad` application on a maze scene as compared to the original optimized C code. Measurements were done on an 8-processor 167-MHz UltraSPARC I.

a provably good work-stealing algorithm [4]. Radiosity, then, is a good test for the capabilities of Cilk's scheduler. Past attempts at parallelizing radiosity have required the algorithm to be modified with explicit load balancing [41].

In order to effectively test Cilk's performance, we prefer not to develop our own radiosity application. It is somewhat "unfair" to develop such a test by writing code that is intentionally designed to work well with Cilk. Also, we would prefer to have a known benchmark against which to measure the parallelized code. Speeding up our own code by parallelizing it is not convincing, because it might be that serial optimizations could perform as well or better. Therefore, a better test is to try to parallelize code that was written and optimized by someone else. If we can speed up code that has already been optimized by graphics experts, our results clearly demonstrate the usefulness of Cilk.

We therefore downloaded a radiosity application, `rad`, which was originally written by Bekaert, Suykens de Laet, and Dutre at the Katholieke Universiteit Leuven in Belgium [2]. The application is large, consisting of 75 source files and around 25,000 lines of code. The application is written in C, and every C program is a legal Cilk program, so "porting" the application to Cilk required no effort.

Correctly parallelizing the code, however, is not as trivial. The code was not originally written to be parallelized. Although we might expect certain operations to be in principle independent, in fact they may use some shared data structures simply because the programmer implemented it that way. Such code would result in data races if those operations were executed in parallel.

Ordinarily, in order to parallelize the code without introducing data races, we would have to search through the entire code looking for shared data structures. The Nondeterminator-2, however, provides an alternate approach. We simply run in parallel those operations that we think should in principle be independent. Then we run the code through the Nondeterminator, which points us to the places in the code where there is unexpected data sharing. We can fix these problems by copying the data, or by adding locks. More importantly, we do not need to examine at all code that is not flagged by the Nondeterminator-2; we simply leave it as is. In that way, we minimize the amount of time we need to spend studying and understanding someone else's code.

When parallelizing the radiosity application, we took precisely this approach of immediately depending on the Nondeterminator-2, although we actually began by using the original Nondeterminator and not the Nondeterminator-2. This particular application was actually developed in conjunction with the Nondeterminator-2, and served as the inspiration for many of that tool's features. We now illustrate some of the details of our effort in order to give a more concrete sense of what was involved.

## The parallelization effort

The first step was to gain an understanding of the underlying radiosity algorithm, so we could figure out what to parallelize. Radiosity is a lighting model that is suited for diffuse environments. Light striking a surface is assumed to undergo an ideal diffuse reflection, meaning that it scatters equally in all directions. The diffuse reflection assumption is in contrast to ray tracing's assumption of specular reflection, wherein a beam of light is assumed to reflect off a surface in another single beam, with the angle of reflection equaling the angle of incidence.

As in many graphics algorithms, radiosity divides the scene into several small "patches." Each patch $i$ has an associated power per unit area $B_i$ from which the color of the patch $i$ can be determined.[1] The idea is that the power leaving patch $i$ is the sum of the power emitted by $i$ (if $i$ is a light source) and the power reflecting off $i$ that comes from all of the other patches in the room. This formula leads to the following set of linear equations [18]:

$$B_i = E_i + \rho_i \Sigma_j B_j F_{ij}$$

where $B_i$ is the power/area of patch $i$, $E_i$ is the emitted power/area of patch $i$, $\rho_i$ is the reflectance of patch $i$, and $F_{ij}$ is the ***formfactor*** from $i$ to $j$, the fraction of radiant power leaving $i$ that arrives at $j$.

We can solve for $B_i$ by numerical iteration. The majority of the calculation time, however, is not spent in the numerical solution, but rather in the calculation of the formfactors $F_{ij}$. The formfactor from a point patch $i$ to a patch $j$ is the fraction of $i$'s hemisphere that $j$ occupies. Computing the formfactor $F_{ij}$ thus requires a double integral over the points of patch $i$ and patch $j$. The formfactors are entirely a property of the geometry of the scene, and do not depend on lighting.

As they are calculated directly from the initial geometry of the scene, distinct formfactors can be computed in parallel. Since the calculation of formfactors comprises the majority of the execution time of the radiosity algorithm, this parallelization should noticeably speed up the entire execution.

Armed with this knowledge, we searched through the `rad` code for the calculation of the formfactors, and ran them in parallel. We then ran the resulting code through

[1] Actually, the color of patches is determined by assigning colors to vertices and then interpolating those colors to the rest of the patch, typically with Gourard shading [19].

the Nondeterminator to look for data races.

Since the code was initially serial, it did not contain any locks, and was therefore abelian. One goal of parallelization is to keep the program abelian "as long as possible," which provides the stronger guarantee for the Nondeterminator-2. (This strategy often amounts to avoiding introducing locks for as long as possible.) When finally forced to make the program nonabelian, the programmer must be sure to think about the implementation more carefully.

The first dag races we ran across in `rad` involved global variables. In some cases, these globals appeared to be used only for convenience, to avoid passing them around as arguments to procedures. We modified the code to pass arguments rather than use globals whenever possible. Another common use of global variables we found was just for statistical purposes, such as timings. These statistics can either be ignored in the parallel execution (i.e. allowed to become garbage values), or they can be updated atomically through the use of locks. Such atomic updates are commutative, and so preserve the abelian property of the program.

The `rad` code does not exactly implement the radiosity algorithm as we have described it. The code does not precompute all the formfactors and then solve the numerical system, as computing all the formfactors would require too much CPU time and memory. Rather, the code interleaves the solution to the system with the formfactor calculations. Specifically, it chooses a single patch $i$ where the error in the $B_i$ approximation is the greatest. It then improves the estimate for $B_i$ by improving its approximation of the formfactors $F_{ij}$ for all other patches $j$. The first few iterations of the application, shown in Figure 10-2, demonstrate how the code interleaves the updating of the patch radiosities with the calculations of the formfactors. This algorithm poses some problems for the parallel execution, because separate iterations of the numerical solution cannot be run in parallel, as each iteration is dependent on the previous one. The calculations of the formfactors from $i$ to all the other patches $j$ can still be parallelized, however.[2]

[2] Once again, we could have rewritten the code to perform more formfactor calculations at once, but then we would have lost the serial optimizations of the original authors.

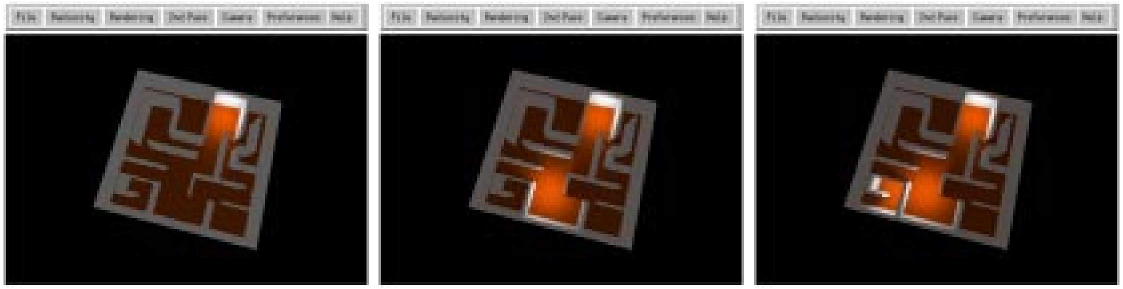


**Figure 10-2**: The first three iterations of the `rad` program on a maze scene. At each iteration, the program refines its formfactor estimates for the patch where the error is greatest. In the first few iterations, the error is greatest near the light sources, so the program appears to be "lighting up" the lights one by one.

The formfactors $F_{ij}$ are stored in a linked list in a data structure for patch $i$. Thus, we encountered a dag race on the updates to this list, as formfactors were being added in parallel to it. Fortunately, the order in which the formfactors occur in the list doesn't matter, so they can be added in parallel as long as the list insertion operations are made to be atomic. We added a lock to each patch data structure for this purpose.

This logic causes the program to be nonabelian, as the order of the nodes in the linked list depends on the order of execution of critical sections. Nonetheless, it is not hard to argue that the Nondeterminator still catches all dag races involving this list. The reason is that the code never reads only part of the list; rather, it always reads the entire list at once. Thus, if any writes race with those reads, they race with the reads regardless of the order of the elements in the list.

The next difficulty we encountered in our parallelization was patch refinement. When the error in the estimate for the formfactor $F_{ij}$ is deemed to be too great, either patch $i$ or $j$ is refined. (The patch refined is the one with the greater surface area.) Refinement means that the patch is subdivided into smaller patches in order to get more accurate radiosity estimates.

If two parallel threads attempt to subdivide a patch $i$ at once, a data race occurs on that patch. It is allowable, however, for either thread to do the actual refinement, as long as the refinement is done only once. This logic can be implemented with locks. The first thread to acquire the "refinement lock" for the patch performs the subdivision, and the second thread waits on that lock. After the first thread finishes the refinement and releases the lock, the second thread acquires the lock but discovers that the refinement has already been done, and so does not repeat it. Also, refining patch $i$ does not destroy $i$, it merely creates "subpatches" of $i$. Therefore, a thread $e_1$ that refines a patch $i$ does not interfere with a parallel thread $e_2$ that calculates directly with $i$. We can thus have many parallel threads calculating formfactors for patch $i$, some of which refine $i$ and some of which do not, without any data races.

This protocol, unfortunately, is entirely nonabelian. A single thread creates and initializes the subpatches of $i$. Many parallel threads read these subpatches, resulting in dag races. These dag races are not actually data races, because the locking protocol assures that no threads read the subpatches until the "first" thread finishes initializing them. This protocol is an example of the "publishing" problem discussed in Chapter 9, and false race reports for this protocol can be avoided by annotating the code with PUBLISH statements.

When a patch is refined, the newly created vertices are added to a list stored in the patch's "surface." A surface is the top-level patch that initially gets created from the scene description. Multiple patches can thus have the same surface, so we need to create a lock for each surface that is acquired when vertices are added to it. Adding vertices to a surface, therefore, is similar to adding formfactors to a patch.

When we initially ran the code, it appeared to be behaving correctly. We later observed, however, that the code was behaving nondeterministically after running for about 10 iterations. Investigation of this problem showed that its manifestation was that a thread would read from freed memory. This discovery led us to think about the mechanism for detecting races with the memory allocator, as discussed in Chapter 9, although that turned out not to be the problem in this particular case.

We had been running the code through the Nondeterminator-2 for only one iteration, expecting that all dag races would show up there. We were also at that point struggling with a large number of false reports (dag races that were not actually data races). This difficulty led us to the idea of a hash table to avoid reporting the "same" race twice, as discussed in Chapter 9. After implementing that idea, we ran the debugging code for many iterations. Many false reports still showed up in the first iteration, but those were not reported again, so that later iterations did not report races. The first new race report appeared in iteration 10, and this report pointed us to the bug that we had seen.

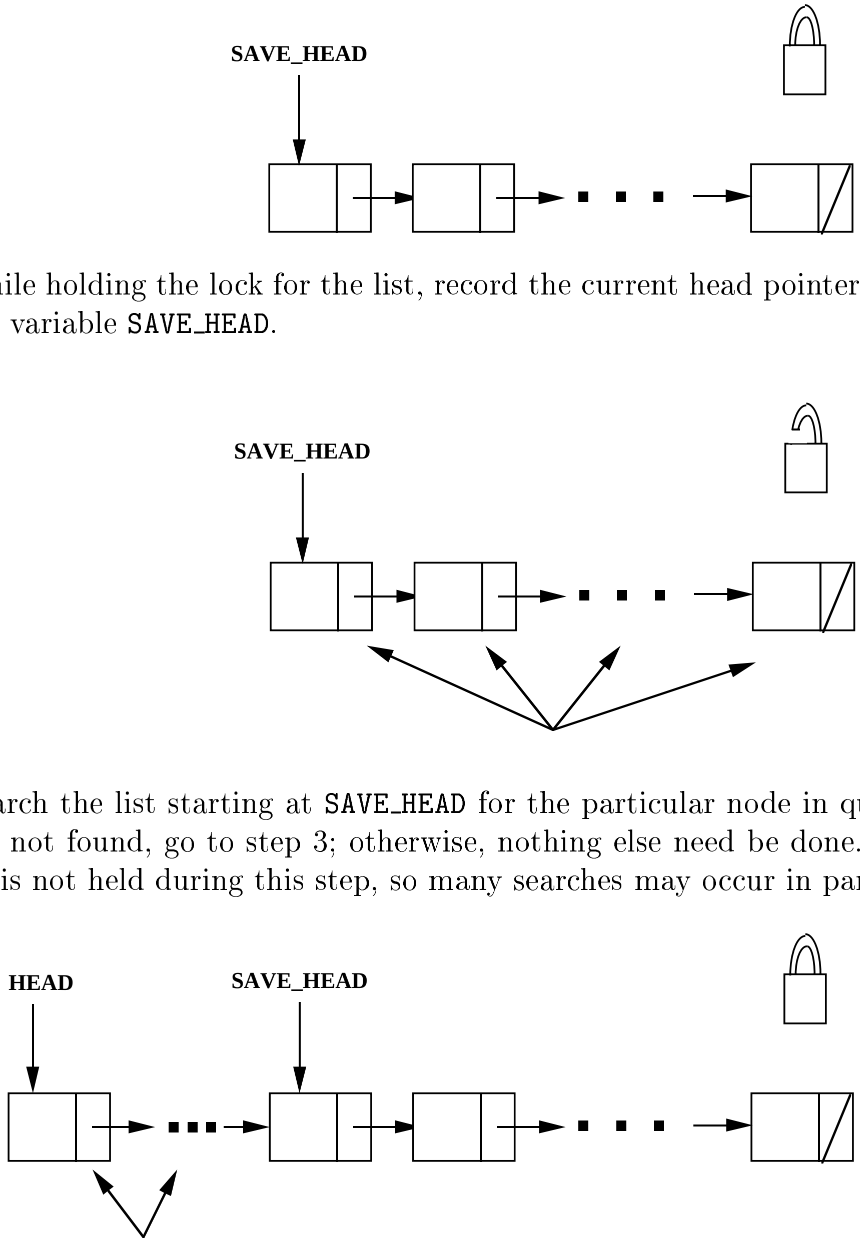
At this point, we were able to obtain a reasonable speedup, but we discovered that the serial code for patch refinement was taking a lot of the execution time. The expensive part of this code is that in order to avoid adding duplicate vertices to the surface's list of vertices, the program must search that list before adding each vertex. This search can be parallelized, because searching only requires reading the elements of the list, not writing them.

This parallelization requires an elaborate protocol, which is described in Figure 10-3. We first obtain the lock for the list, and record the head pointer of the list. We then release the lock and search the rest of the list for the vertex in question. If the vertex is found, then we don't need to add it to the list, so we're finished. If the vertex is not found, then we acquire the lock for the list again in order to add it. Other vertices, however, may have been added to the list since the time we began our search. We thus must search the beginning of the list, up to the point where we began our earlier search, in order to check if a parallel thread has already added the vertex in question. If not, then we add the vertex to the front of the list while still holding the lock for the list. The idea of this protocol is that the majority of the computation time is spent searching the bulk of the list with no locks held, which can be done by many threads in parallel.[3]

This protocol is once again nonabelian. When vertices are added to the list, they are being "published." False races can thus be avoided by judicious use of PUBLISH statements.

As mentioned above, a "refinement lock" is acquired when a patch is refined. The parallelism within patch refinement, therefore, actually occurs while a lock is held. This behavior is in theory disallowed, but in reality it does not cause any fatal

[3] This code could likely be improved by using a more efficient data structure than a linked list, but we do not wish to change the underlying algorithms of the original implementation.



Step 1. While holding the lock for the list, record the current head pointer of the list into a local variable `SAVE_HEAD`.

Step 2. Search the list starting at `SAVE_HEAD` for the particular node in question. If the node is not found, go to step 3; otherwise, nothing else need be done. The lock for the list is not held during this step, so many searches may occur in parallel.

Step 3. Acquire the lock for the list again. Search for the node in question from the current head pointer of the list until the node saved in `SAVE_HEAD`. If the node is not found, add it to the front of the list.

**Figure 10-3**: The protocol for adding vertices to the surface's vertex list. Most of the searching of the list can be done in parallel.

problems. The only danger is that the Nondeterminator-2 may miss races because the refinement lock appears to be protecting the parallel accesses when in fact it does not. In this particular case, no races are missed, because there are parallel threads that operate on different patches but the same surface; these parallel accesses are not protected by any single patch refinement lock.

When we ran the `rad` application on many processors, we discovered that the form-factor calculations and patch refinement were sufficiently fast that other portions of the code were becoming bottlenecks. We parallelized two other CPU intensive routines. One of these routines calculates the color of vertices from the patch radiosities; this color calculation can be done for the entire scene in parallel. The other routine performs "T-vertex elimination," which essentially deallocates memory for certain undesirable kinds of vertices.

## Parallelization results

Timings of the `rad` routines are given in Figure 10-4. As expected, the formfactor/patch refinement calculations dominate execution time in the one processor execution. Vertex color computation and T-vertex elimination also comprise a sizeable portion of the execution time. The rest of the CPU time is labeled "Other," and corresponds to the remaining code, which was not parallelized. This code includes, for example, the numerical iterations updating the radiosity values and the hardware rendering of the scene to the monitor. As the parallel routines speed up in multiprocessor execution, the formfactor calculation with patch refinement is still the most expensive operation, but the time spent in nonparallelized code becomes comparable.

Figure 10-5 shows these measurements as speedups as compared to the original optimized C code. In particular, we observe that the one processor Cilk version is negligibly slower than the original C version.[4] The speedup curve for the entire parallelized application shows the combination of the running times of the four components given in Figure 10-4. The entire execution achieves a 5.97 times speedup on

[4] The added overhead of Cilk procedure calls is balanced by the speedup from Cilk's fast memory allocator.

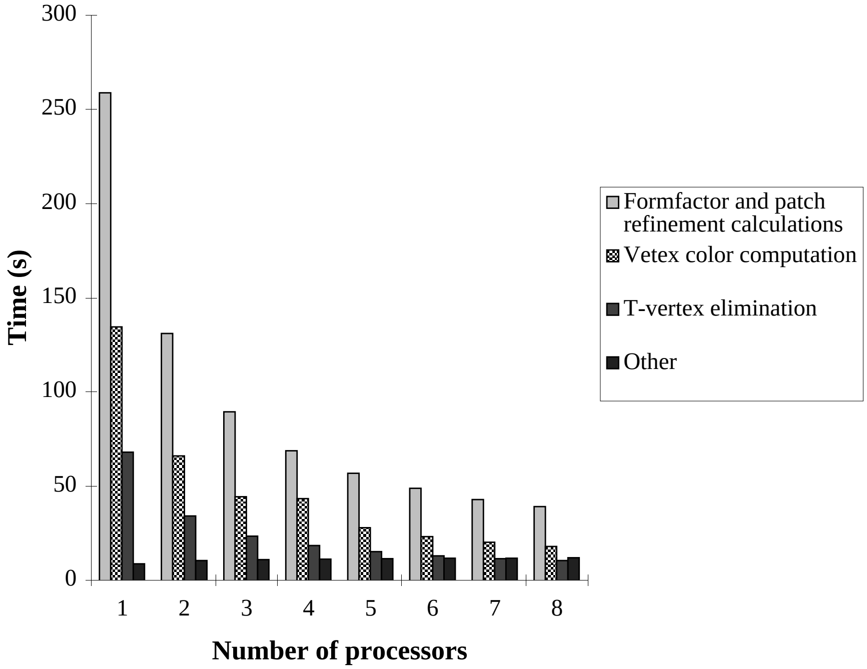


**Figure 10-4**: Running times of the components of the `rad` application. Timings were done of 100 iterations of the application on the maze scene on an 8-processor 167-MHz UltraSPARC I.

8 processors.

Additionally, Cilk provides a way to measure the work and critical path of the computation. The ***work*** $T_1$ is the time it takes the Cilk program to execute on a single processor. The ***critical path*** $T_\infty$ is the time it would take to execute the program on infinitely many processors. The ***average parallelism*** is defined to be $T_1/T_\infty$, and represents a measure of the speedup that the program can obtain. When the average parallelism of the program is much greater than the number of processors $P$ being used, a theorem shows that Cilk's scheduler runs the program in time approximately $T_1/P$ with high probability [4]. The average parallelism of the formfactor calculations is measured as 221. Unfortunately, this measurement does not account for time spent in contention for user locks; such contention both adds work for the program and reduces parallelism. On 8 processors, however, the work is only increased by 18 percent, and the average parallelism is around 195. This high average parallelism implies that the calculations could be further sped up with more than 8 processors.

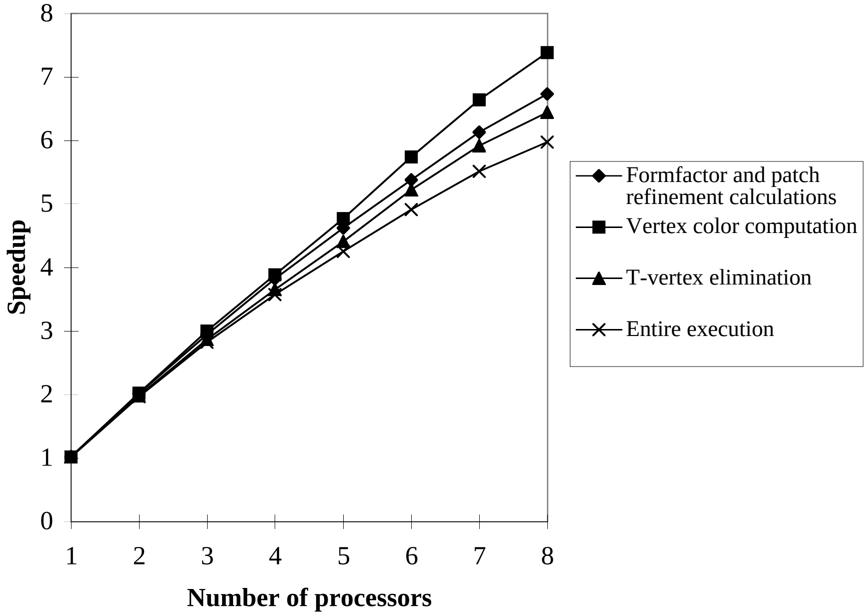


**Figure 10-5**: Speedup of the components of the `rad` application on the maze scene as compared to the original optimized C code. The measurements were taken on an 8-processor 167-MHz UltraSPARC I.

Upon examination, we find that our parallelization changed less than 5 percent of the total code. We were not required to examine nor understand the majority of the code. Furthermore, the Nondeterminator-2 gives us reason to believe that the code is data-race free. The combination of Cilk and the Nondeterminator-2 made it practical to efficiently and correctly parallelize this large-scale, real-world application.

# Chapter 11

# Conclusion

The many challenges to successfully parallelizing programs include expressing the parallelism in the program, getting good performance out of parallel hardware, and debugging. Cilk is designed to address the first two issues, and in this thesis, we have addressed the third. We presented the All-Sets and Brelly algorithms for finding dag races, and explained how those races relate to the semantics of the program. We showed how these tools were used to parallelize a large, real-world radiosity application.

Although the Nondeterminator-2 is an efficient tool for race-detection, many issues surrounding its use remain unresolved. A key decision by Cilk programmers is whether to adopt the umbrella locking discipline. Programmers might first debug with All-Sets, but unless they have adopted the umbrella discipline, they will be unable to fall back on Brelly if All-Sets seems too slow. We recommend that programmers use the umbrella discipline initially, which is good programming practice in any event, and only use All-Sets if they are forced to drop the discipline.

Even when using All-Sets, users can encounter false positives and false negatives from the Nondeterminator-2 when their programs are nonabelian. It is an open question whether there are other classes of programs (besides abelian programs) for which the Nondeterminator-2 can provide guarantees of determinacy. If we examine the proof in Chapter 8, we find that we don't actually need the strong requirement of commutativity that each of two critical sections must execute the "identical sequence of instructions on the identical memory locations" in either order of execution. Rather, it is only necessary that each critical section read and write the same set of memory locations in either execution order, and also that in either execution order each critical section acquire the same locks in the same order. Thus, we may be able to consider as abelian some programs that are not formally abelian by the definition given in Chapter 8.

This generalization of the definition of abelian has implications for nonabelian programs as well; it could provide an approach to avoid some of the false negative problems discussed in Chapter 7. It may be possible for a compiler to conservatively estimate the memory locations touched by critical sections. Thus, even if a critical section does not happen to touch all of those locations in a given computation, we may be able to find dag races in other computations using those conservative estimates.

In Chapter 10, we argued that although the process of adding nodes to a linked list in parallel is nonabelian, in practice the Nondeterminator-2 does not miss races, because the order of the nodes in the linked list doesn't matter. It may be possible to prove such a claim by proving that the code operates on the same set of memory locations regardless of the order of the nodes in the linked list.

The techniques we have presented for reducing the number of false race reports in nonabelian programs are at best imperfect. It would be preferable to have a "higher level" language construct for annotating code than PUBLISH, which requires the user to be explicitly aware of the exact memory locations being published. Furthermore, in some cases PUBLISH does not properly convey the semantics of the user's code. The user may in fact be using critical sections to synchronize the entire program, and not to publish any particular memory. Such semantics might be better handled by introducing other language constructs into Cilk that precisely express the synchronization semantics intended. A preferable solution is probably to once again allow the user to annotate the code, expressing the fact that certain critical regions actually synchronize the program. In order to properly handle such directives, we need to extend the SP-BAGS algorithm to graphs that are not series-parallel.

Missed races and false reports are not a problem when the program being debugged is abelian, but programmers would like to know whether an ostensibly abelian program is actually abelian. Dinning and Schonberg give a conservative compile-time algorithm to check if a program is “internally deterministic” [11], and we have given thought as to how the abelian property might likewise be conservatively checked. The parallelizing compiler techniques of Rinard and Diniz [38] may be applicable.

The guarantee of the Nondeterminator-2 for abelian programs requires that the program be deadlock-free, which is left to the user to verify. We would prefer to have a way of checking if a program, or at least a computation of a program, is deadlock free. While this problem in general appears difficult, there may be a reasonable, flexible locking discipline that precludes deadlocks and that allows efficient detection.

Although we believe that the Nondeterminator-2 is a useful tool, we have the unfair advantage of having developed it. Other programmers may not want to take the time to learn how to use the tool. Past experience has shown that many programmers assume that their program is correct if they run it several times without failures. Will such programmers be willing to try out a debugging tool that may only produce false race reports anyway? The answer remains to be seen, but from our experience we know that correct parallelization is hard, and we believe that any user would be well advised to take the time to learn how to debug with the Nondeterminator-2.

# Appendix A

# Deadlock in the Computation

In this appendix, we give a proof of Lemma 17.[1] This lemma shows that for abelian programs, a deadlock in a dag-race free computation corresponds exactly to a deadlock in the program.

Ideally, we would have an algorithm that checks for deadlocks in a computation dag. Users would run this algorithm along with All-Sets or Brelly to directly use the results of Theorems 15 and 16. Since we do not (yet) have an efficient algorithm to detect deadlocks in a dag, however, using Theorems 15 and 16 directly requires users to manually examine computation dags for deadlocks. Users, however, presumably don't really care about deadlocks in computations; they care whether their programs can actually deadlock. Fortunately, Lemma 17 shows that checking an abelian program for deadlocks is equivalent to checking any dag-race free computation of that program for deadlocks.

In our current formulation, proving that a deadlock scheduling of a computation is feasible is not sufficient to show that the machine actually deadlocks. A deadlock scheduling is one that cannot be extended in the computation, but it may be possible for the machine to extend the execution if the next machine instruction does not correspond to one of the possibilities in the dag. In this appendix, in order to prove machine deadlocks, we think of a lock instruction as being composed of

[1] This proof is joint work with Keith Randall.

two instructions: LOCK_ATTEMPT and LOCK_SUCCEED. Every two LOCK_SUCCEED instantiations that acquire the same lock must be separated by an UNLOCK of that lock, but multiple LOCK_ATTEMPT instantiations for the same lock can be executed by different interpreters in arbitrary order. In other words, LOCK_ATTEMPT instructions can always be executed by the interpreter, but LOCK_SUCCEED instructions cannot be executed unless no other interpreter holds the lock. If an interpreter executes a LOCK_ATTEMPT instruction, the next instruction executed by the interpreter must be a LOCK_SUCCEED instruction for the same lock. A feasible deadlock scheduling is therefore an actual machine deadlock, because the LOCK_SUCCEED instantiations that come next in the dag are always the same as the next possible instantiations for the machine.

A LOCK_ATTEMPT instantiation commutes with any other parallel instantiation. For convenience, we still use the single instantiation LOCK to mean the sequence LOCK_ATTEMPT LOCK_SUCCEED.

It is the proof of Lemma 17 that requires the extra technical condition on commutativity that is mentioned in Chapter 8, which we call ***prefix commutativity***: essentially, a prefix of a region locked by the same lock as a complete region must "commute" with the complete region. Precisely, given a partial scheduling $X$, two parallel regions $R_1$ and $R_2$ that are surrounded by the same lock, and $R_2'$ a prefix of $R_2$, then $XR_1R_2'$ being feasible implies that $XR_2'$ is feasible. The reason for this requirement is that it may be the case that it is never possible for two complete regions to execute adjacent to each other. An example is shown in Figure A-1. In that program, it is never possible for the two regions that lock the lock B (lines 11–17 and 20–26) to execute adjacent to each other, because those regions each acquire locks that are held by the other thread. Therefore, without the requirement of prefix commutativity those regions would not be required to commute in any way. It is possible, however, to execute one entire region, say lines 11–17, and then a prefix of the other, namely from line 20 up to the LOCK_ATTEMPT(A) in line 23. Prefix commutativity requires that this prefix consist of the same instantiations as if it were executed before the complete region in lines 11–17. The code for the program in Figure A-1 does not

```
    int x;
    Cilk_lockvar A;
    Cilk_lockvar B;
    Cilk_lockvar C;

    cilk double main()
    {
1:    x = 0;
2:    Cilk_lock_init(A);
3:    Cilk_lock_init(B);
4:    Cilk_lock_init(C);
5:    spawn foo1();
6:    spawn foo2();
7:    sync;
8:    printf("%d", x);
9:    return 0;
    }
```

```
     cilk void foo1()
     {
10:    Cilk_lock(A);
11:    Cilk_lock(B);
12:    x++;
13:    if (x==2)
         {
14:      Cilk_lock(C);
15:      x = 5;
16:      Cilk_unlock(C);
         }
17:    Cilk_unlock(B);
18:    Cilk_unlock(A);
     }
```

```
     cilk void foo2()
     {
19:    Cilk_lock(C);
20:    Cilk_lock(B);
21:    x++;
22:    if (x==2)
         {
23:      Cilk_lock(A);
24:      x = 6;
25:      Cilk_unlock(A);
         }
26:    Cilk_unlock(B);
27:    Cilk_unlock(C);
     }
```

**Figure A-1**: A program that illustrates the need for the prefix commutativity requirement. The program does not deadlock; of the two LOCK(B) · · · UNLOCK(B) regions (lines 11–17 and 20–26), only the second one to execute acquires another lock (either A or C). Furthermore, those regions can never execute entirely adjacent to each other, for the second one to execute must wait for the entire other thread to complete. This program does not have any data (or dag) races, but it may produce a final value of `x` as either 5 or 6. The prefix commutativity requirement means that this program is not considered to be abelian, because the prefix in lines 20–23 does not "commute" with the complete region in lines 11–17.

satisfy this requirement, and so the program is not abelian. In particular, we observe that the program uses special logic to avoid the possibility of deadlock. The prefix commutativity requirement allows us to prove that when parallel regions cannot actually occur adjacent in an execution, then the program must contain a deadlock.

To prove Lemma 17, we first introduce new versions of Lemmas 11, 12, and 13 that assume a deadlock-free program instead of a deadlock-free dag. We then use these modified versions to prove Lemma 17.

**Lemma 20 (Reordering)** *Let $G$ be a dag-race free computation resulting from the execution of a deadlock-free abelian program, and let $R_1$ and $R_2$ be two parallel regions in $G$. Then:*

1. *Let $X$ be a partial scheduling of $G$ of the form $X_1R_1R_2X_2$. The partial scheduling $X$ and the partial scheduling $X_1R_2R_1X_2$ are equivalent.*

2. *Let $Y$ be a feasible partial scheduling of $G$ of the form $Y = Y_1R_1R_2'$, where $R_2'$*

*is a prefix of $R_2$. Then then the partial scheduling $Y_1 R'_2$ is feasible.*

*Proof:* We prove the lemma by double induction on the nesting count of the regions. Our inductive hypothesis is the theorem as stated for regions $R_1$ of nesting count $i$ and regions $R_2$ of nesting count $j$. The proofs for part 1 and part 2 are similar, so sometimes we will prove part 1 and provide the modifications needed for part 2 in parentheses.

Base case: $i = 0$. Then $R_1$ is a single instantiation. Since $R_1$ and $R_2$ ($R'_2$) are parallel and are adjacent in $X$ ($Y$), no instantiation of $R_2$ ($R'_2$) can be guarded by a lock that guards $R_1$, because any lock held at $R_1$ is not released until after $R_2$ ($R'_2$). Therefore, since $G$ is dag-race free, either $R_1$ and $R_2$ ($R'_2$) access different memory locations or $R_1$ is a READ and $R_2$ ($R'_2$) does not write to the location read by $R_1$. In either case, the instantiations of each of $R_1$ and $R_2$ ($R'_2$) do not affect the behavior of the other, so they can be executed in either order without affecting the final memory state.

Base case: $j = 0$. Symmetric with above.

Inductive step: In general, $R_1$ has nesting count $i \geq 1$, and is of the form LOCK(A) $\cdots$ UNLOCK(A). $R_2$ of count $j \geq 1$ has the form LOCK(B) $\cdots$ UNLOCK(B). If A $=$ B, then $R_1$ and $R_2$ commute by the definition of abelian. Part 1 then follows from the definition of commutativity, and part 2 follows from prefix commutativity. Otherwise, there are three possible cases.

Case 1: Lock A does not appear in $R_2$ ($R'_2$). For part 1, we start with the sequence $X_1 R_1 R_2 X_2$ and commute pieces of $R_1$ one at a time with $R_2$: first, the instantiation UNLOCK(A), then the immediate subregions of $R_1$, and finally the instantiation LOCK(A). The instantiations LOCK(A) and UNLOCK(A) commute with $R_2$, because A does not appear anywhere in $R_2$. Each subregion of $R_1$ commutes with $R_2$ by the inductive hypothesis, because each subregion has lower nesting count than $R_1$. After commuting all of $R_1$ past $R_2$, we have an equivalent execution $X_1 R_2 R_1 X_2$. For part 2, the same procedure can be used to drop pieces of $R_1$ in the feasible partial schedule $Y_1 R_1 R'_2$ until the feasible partial schedule $Y_1 R'_2$ is reached.

Case 2: Lock B does not appear in $R_1$. The argument for part 1 is symmetric with

Case 1. For part 2, we break $R_2'$ into its constituents: $R_2' = \text{LOCK}(\text{B})R_{2,1} \ldots R_{2,n}R_2''$, where $R_{2,1}$ through $R_{2,n}$ are complete regions, and $R_2''$ is a prefix of a region. The instantiation LOCK(B) commutes with $R_1$ because B does not appear in $R_1$, and the complete regions $R_{2,1}$ through $R_{2,n}$ commute with $R_1$ by induction. From the schedule $Y_1\text{LOCK}(\text{B})R_{2,1} \ldots R_{2,n}R_1R_2''$, we again apply the inductive hypothesis to drop $R_1$, which proves that $Y_1\text{LOCK}(\text{B})R_{2,1} \ldots R_{2,n}R_2'' = Y_1R_2'$ is feasible.

Case 3: Lock A appears in $R_2$ $(R_2')$, and lock B appears in $R_1$. For part 1, if both schedulings $X_1R_1R_2X_2$ and $X_1R_2R_1X_2$ are infeasible, then we are done. Otherwise, we prove a contradiction by showing that the program can deadlock. Without loss of generality, let the scheduling $X_1R_1R_2X_2$ be a feasible scheduling. Because $X_1R_1R_2X_2$ is a feasible scheduling, the partial scheduling $X_1R_1R_2$ is feasible as well.

We now continue the proof for both parts of the lemma. Let $\alpha_1$ be the prefix of $R_1$ up to (and including) the first LOCK_ATTEMPT(B) instantiation, let $\beta_1$ be the rest of $R_1$, and let $\alpha_2$ be the prefix of $R_2$ $(R_2')$ up to (and including) the first LOCK_ATTEMPT of a lock acquired in $R_2$ $(R_2')$ that is acquired but not released in $\alpha_1$. At least one such lock exists, namely A, so $\alpha_2$ is not all of $R_2$ $(R_2')$.

We show that the partial scheduling $X_1\alpha_1\alpha_2$ is also feasible. This partial scheduling, however, cannot be completed to a full scheduling of the program because $\alpha_1$ and $\alpha_2$ each hold the lock that the other is attempting to acquire.

We prove the partial scheduling $X_1\alpha_1\alpha_2$ is feasible by starting with the feasible partial scheduling $X_1R_1\alpha_2 = X_1\alpha_1\beta_1\alpha_2$ and dropping complete subregions and unpaired unlocks in $\beta_1$ from in front of $\alpha_2$. The sequence $\beta_1$ has two types of instantiations, those in regions completely contained in $\beta_1$, and unpaired unlocks.

Unpaired unlocks in $\beta_1$ must have their matching lock in $\alpha_1$, so that lock does not appear in $\alpha_2$ by construction. Therefore, an unlock instantiation just before $\alpha_2$ commutes with $\alpha_2$ and thus can be dropped from the schedule. Any complete region just before $\alpha_2$ can be dropped by the inductive hypothesis. When we have dropped all instantiations in $\beta_1$, we obtain the feasible partial scheduling $X_1\alpha_1\alpha_2$ which cannot be completed, and hence the program has a deadlock. ■

**Lemma 21 (Region grouping)** *Let $G$ be a dag-race free computation generated by the execution of a deadlock-free abelian program. Let $X_1 X X_2$ be a scheduling of $G$, for some instantiation sequences $X_1$, $X$, and $X_2$. Then, there exists an instantiation sequence $X'$ such that $X_1 X' X_2$ is equivalent to $X_1 X X_2$ and every region entirely contained in $X'$ is contiguous.*

*Proof:* As a first step, we create $X''$ by commuting each LOCK_ATTEMPT in $X$ to immediately before the corresponding LOCK_SUCCEED. In this way, every complete region begins with a LOCK instantiation. If there is no corresponding LOCK_SUCCEED in $X$, we commute the LOCK_ATTEMPT instantiation to the end of $X''$.

Next, we create our desired $X'$ by grouping all the complete regions in $X''$ one at a time. This can be done using identical techniques to the proof of Lemma 12, applying Lemma 20 in place of Lemma 11. ■

**Lemma 22** *Let $G$ be a dag-race free computation resulting from the execution of a deadlock-free abelian program. Then every legal scheduling of $G$ is feasible and yields the same final memory state.*

*Proof:* The proof is identical to the proof of Lemma 13, using the Reordering and Region Grouping lemmas from this appendix in place of those from Chapter 8. ■

We restate and then prove Lemma 17.

**Lemma 17** *Let $G$ be a dag-race free computation generated by an abelian program. $G$ is deadlock free if and only if the program is deadlock free (on the same input).*

*Proof:* ($\Leftarrow$) If $G$ is deadlock free, then every machine execution of the program is a scheduling of $G$ by Lemma 14, so the machine cannot have a deadlock execution.

($\Rightarrow$) By contradiction. Assume that a deadlock-free abelian program $P$ can generate a dag-race free computation $G$ that has a deadlock. We show that $P$ can deadlock, which is a contradiction.

The proof has two parts. In the first part, we generate a feasible scheduling $Y$ of $G$ that is "almost" a deadlock scheduling. Then, we show that $Y$ can be modified slightly to generate a deadlock scheduling that is also feasible, which proves the contradiction.

Every deadlock scheduling contains a set of threads $e_1, e_2, \ldots e_n$, some of which are completed and some of which are not. Each thread $e_i$ has a ***depth***, which is the length of the longest path in $G$ from the initial node to the last instantiation in $e_i$. We can define the ***depth*** of a deadlock scheduling as the $n$-tuple $\langle \mathit{depth}(e_1), \mathit{depth}(e_2), \ldots, \mathit{depth}(e_n)\rangle$, where we order the $e_i$ such that $\mathit{depth}(e_1) \geq \mathit{depth}(e_2) \geq \ldots \geq \mathit{depth}(e_n)$. Depths of deadlocked schedulings are compared in the dictionary order.[2]

We generate the scheduling $Y$ of $G$ which is almost a deadlock scheduling by modifying a particular deadlock scheduling of $G$. We choose the deadlock scheduling $X$ from which we will create the scheduling $Y$ to have the maximum depth of any deadlock scheduling of $G$.

Let us examine the structure of $X$ in relation to $G$. The deadlock scheduling $X$ divides $G$ into a set of completely executed threads, $X_1$, a set of unexecuted threads $X_2$, and a set of partially executed threads $T = \{t_1, \ldots, t_n\}$, which are the threads whose last executed instantiation in the deadlock scheduling is a LOCK_ATTEMPT. We divide each of the threads in $T$ into two pieces. Let $A = \{\alpha_1, \ldots, \alpha_n\}$ be the parts of the $t_i$ up to and including the last executed instantiation, and let $B = \{\beta_1, \ldots, \beta_n\}$ be the rest of the instantiations of the $t_i$. We say that $\alpha_i$ ***blocks*** $\beta_j$ if the first instantiation in $\beta_j$ is a LOCK_SUCCEED on a lock that is acquired but not released by $\alpha_i$.

$X$ is a deadlock scheduling containing the instantiations in $X_1 \cup A$. To isolate the effect of the incomplete regions in $A$, we construct the legal scheduling $X'$ which first schedules all of the instantiations in $X_1$ in the same order as they appear in $X$, and then all of the instantiations in $A$ in the same order as they appear in $X$.

The first instantiations of the $\beta_i$ cannot be scheduled in $X'$ because they blocked

[2] The dictionary order $<_D$ is a partial order on tuples that can be defined as follows: The size 0 tuple is less than any other tuple. $\langle i_i, i_2, \ldots, i_m\rangle <_D \langle j_1, j_2, \ldots, j_n\rangle$ if $i_1 < j_1$ or if $i_1 = j_1$ and $\langle i_2, i_3, \ldots, i_m\rangle <_D \langle j_2, j_3, \ldots, j_n\rangle$.

by some $\alpha_j$. We now prove that the blocking relation is a bijection. Certainly, a particular $\beta_i$ can only be blocked by one $\alpha_j$. Suppose there exists an $\alpha_j$ blocking two or more threads in $B$. Then by the pigeonhole principle some thread $\alpha_k$ blocks no threads in $B$. This contradicts that fact that $X$ has maximum depth, because the deadlock scheduling obtained by scheduling the sequence $X_1 t_k$, all subsequently runnable threads in $X_2$ in any order, and then the $n-1$ partial threads in $A - \{\alpha_k\}$ is a deadlock scheduling with a greater depth than $X$.

Without loss of generality, let $\alpha_2$ be a thread in $A$ with a deepest last instantiation. Since the blocking relation is a bijection, only one thread blocks $\beta_2$; without loss of generality, let it be $\alpha_1$. Break $\alpha_1$ up into two parts, $\alpha_1 = \alpha_1^L \alpha_1^R$, where the first instantiation of $\alpha_1^R$ attempts to acquire the lock that blocks $\beta_2$. ($\alpha_1^L$ may be empty.) To construct a legal schedule, we start with $X'$ and remove the instantiations in $\alpha_1^R$ from $X'$. The result is still a legal scheduling because we did not remove any UNLOCK without also removing its matching LOCK. We then schedule the first instantiation of $\beta_2$, which we know is legal because we just unblocked it. We then complete the scheduling of the threads in $T$ by scheduling the remaining instantiations in $T$ ($\alpha_1^R$ and all instantiations in $B$ except for the first one in $\beta_2$). We know that such a scheduling exists, because if it didn't, then there would be a deeper deadlock schedule (because we executed one additional instantiation from $\beta_2$, the deepest incomplete thread, and we didn't remove any completed threads). We finish off this legal scheduling by completing $X_2$ in topological sort order.

As a result, the constructed schedule consists of four pieces, which we call $Y_1$, $Y_2$, $Y_3'$, and $Y_4$. The sequence $Y_1$ is some scheduling of the instantiations in $X_1$, $Y_2$ is some scheduling of the instantiations in $\alpha_1^L \cup \alpha_2 \cup \ldots \cup \alpha_n$, $Y_3'$ is some scheduling of the instantiations in $\alpha_1^R \cup \beta_1 \cup \ldots \cup \beta_n$, and $Y_4$ is some scheduling of the instantiations in $X_2$. To construct $Y$, we first group the complete regions in $Y_3'$ using Lemma 21 to get $Y_3$, and then define $Y$ to be the schedule $Y_1 Y_2 Y_3 Y_4$. Since $Y$ is a (complete) scheduling of $G$, it is feasible by Lemma 22.

The feasible scheduling $Y$ is almost the same as the deadlock scheduling $X$, except $\alpha_1^R$ is not in the right place. We further subdivide $\alpha_1^R$ into two pieces, $\alpha_1^R = \alpha_1' \alpha_1''$, where $\alpha_1'$ is the maximum prefix of $\alpha_1^R$ that contains no LOCK_SUCCEED instantiations of locks that are held but not released by the instantiations in $\alpha_1^L, \alpha_2, \ldots, \alpha_n$. (Such an $\alpha_1'$ must exist in $\alpha_1^R$ by choice of $\alpha_1^R$, and furthermore $\alpha_1'$ is contiguous in $Y$ because $\beta_1$ completes the region started at $\alpha_1'$, and both $\beta_1$ and $\alpha_1'$ are part of $Y_3$.) We now drop all instantiations after $\alpha_1'$ to make a partial scheduling. We then commute $\alpha_1'$ to the beginning of $Y_3$, dropping instantiations as we go, to form the feasible scheduling $Y_1 Y_2 \alpha_1'$. Two types of instantiations are in front of $\alpha_1'$. Complete regions before $\alpha_1'$ are contiguous and can be dropped using Lemma 20. Unlock instantiations can be dropped from in front of $\alpha_1'$ because they are unlocks of some lock acquired in $\alpha_1^L, \alpha_2, \ldots, \alpha_n$, which do not appear in $\alpha_1'$ by construction. By dropping instantiations, we arrive at the feasible scheduling $Y_1 Y_2 \alpha_1'$, which is a deadlock scheduling, as every thread is blocked. This completes the proof. ■

# Bibliography

[1] J. Barnes and P. Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, 1986.

[2] Philippe Bekaert, Frank Suykens de Laet, and Philip Dutre. Renderpark, 1997. Available on the Internet from `http://www.cs.kuleuven.ac./cwis/research/graphics/RENDERPARK/`.

[3] Robert D. Blumofe. *Executing Multithreaded Programs Efficiently.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1995.

[4] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 1996.

[5] Guang-Ien Cheng. Algorithms for race detection in multithreaded programs with atomicity. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1998.

[6] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith Randall, and Andrew F. Stark. Detecting data races in cilk programs that use locks. In *In proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Puerto Vallarta, Mexico, June 28–July 2 1998. To appear.

[7] Cilk-5.1 Reference Manual. Available on the Internet from `http://theory.lcs.mit.edu/~cilk`.

[8] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachusetts, 1990.

[9] David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 164–175, Santa Clara, California, April 1991.

[10] Anne Dinning and Edith Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)*, pages 1–10. ACM Press, 1990.

[11] Anne Dinning and Edith Schonberg. Detecting access anomalies in programs with critical sections. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 85–96. ACM Press, May 1991.

[12] Perry A. Emrath, Sanjoy Ghosh, and David A. Padua. Event synchronization analysis for debugging parallel programs. In *Proceedings of Supercomputing '91*, pages 580–588, November 1991.

[13] Perry A. Emrath and Davis A. Padua. Automatic detection of nondeterminacy in parallel programs. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, pages 89–99, Madison, Wisconsin, May 1988.

[14] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–11, Newport, Rhode Island, June 1997.

[15] Yaacov Fenster. Detecting parallel access anomalies. Master's thesis, Hebrew University, March 1998.

[16] Matteo Frigo, Keith H. Randall, and Charles E. Leiserson. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*, Montreal, Canada, June 1998. To appear.

[17] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum flow problem. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pages 136–146, Berkeley, California, 28–30 May 1986.

[18] Cindy M. Goral, Kenneth K. Torrance, Donald P. Greenberg, and Bennett Battaile. Modeling the interaction of light between diffuse surfaces. *Computer Graphics*, 18(3):213–222, July 1984.

[19] Henri Gourard. *Computer display of curved surfaces*. PhD thesis, University of Utah, 1971.

[20] Michael Halbherr, Yuli Zhou, and Chris F. Joerg. MIMD-style parallel programming with continuation-passing threads. In *Proceedings of the 2nd International Workshop on Massive Parallelism: Hardware, Software, and Applications*, Capri, Italy, September 1994.

[21] David P. Helmbold, Charles E. McDowell, and Jian-Zhong Wang. Analyzing traces with anonymous synchronization. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages II70–II77, August 1990.

[22] Richard C. Holt. Some deadlock properties of computer systems. *Computing Surveys*, 4(3):179–196, September 1972.

[23] Christopher F. Joerg. *The Cilk System for Parallel Multithreaded Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, January 1996.

[24] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, second edition, 1988.

[25] Philip N. Klein, Hsueh-I Lu, and Robert H. B. Netzer. Race-condition detection in parallel computation with semaphores. Technical Report CS-96-04, Brown University, January 1996.

[26] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[27] Leslie Lamport. The mutual exclusion problem: Part I — A theory of interprocess communication. *Journal of the Association for Computing Machinery*, 33(2):313–326, April 1986.

[28] John Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of Supercomputing'91*, pages 24–33. IEEE Computer Society Press, 1991.

[29] Barton P. Miller and Jong-Deok Choi. A mechanism for efficient debugging of parallel programs. In *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 135–144, Atlanta, Georgia, June 1988.

[30] Sang Lyul Min and Jong-Deok Choi. An efficient cache-based access anomaly detection scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 235–244, Palo Alto, California, April 1991.

[31] Greg Nelson, K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Extended static checking home page, 1996. Available on the Internet from `http://www.research.digital.com/SRC/esc/Esc.html`.

[32] Robert H. B. Netzer and Sanjoy Ghosh. Efficient race condition detection for shared-memory programs with Post/Wait synchronization. In *Proceedings of the 1992 International Conference on Parallel Processing*, St. Charles, Illinois, August 1992.

[33] Robert H. B. Netzer and Barton P. Miller. Improving the accuracy of data race detection. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)*, pages 133–144, April 1991.

[34] Robert H. B. Netzer and Barton P. Miller. Experience with techniques for refining data race detection. Technical Report CS-92-55, Brown University, November 1992.

[35] Robert H. B. Netzer and Barton P. Miller. What are race conditions? *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992.

[36] Itzhak Nudler and Larry Rudolph. Tools for the efficient development of efficient parallel programs. In *Proceedings of the First Israeli Conference on Computer Systems Engineering*, May 1986.

[37] Dejan Perković and Peter Keleher. Online data-race detection via coherency guarantees. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, Washington, October 1996.

[38] Martin C. Rinard and Pedro C. Diniz. Commutativity analysis: A new analysis framework for parallelizing compilers. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 54–67, Philadelphia, Pennsylvania, May 1996.

[39] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic race detector for multi-threaded programs. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP)*, October 1997.

[40] Volker Strumpen. Compiler technology for portable checkpoints, 1998. Extended abstract, available at `http://theory.lcs.mit.edu/~strumpen/porch.ps.gz`.

[41] D. Stuttard, A. Worral, D.J. Paddon, and C.P. Willis. A parallel radiosity system for large data sets. In *ACM International Conference on Computer Graphics*, Pilzen, 1995.

[42] Sunsoft. lock_lint, 1994. See Sun Workshop Documentation for command-line utilities, available at `http://www.math.colostate.edu/manuals/sunpro/workshop/command_line/WS_locklint.doc.html`.

[43] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the Association for Computing Machinery*, 22(2):215–225, April 1975.