Algorithms for Data-Race Detection in Multithreaded Programs

by

Guang-Ien Cheng

S.B., Computer Science and Engineering
Massachusetts Institute of Technology, 1997

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

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

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

Algorithms for Data-Race Detection in Multithreaded Programs

by

Guang-Ien Cheng

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

Abstract

If two parallel threads access the same location and at least one of them performs a write, a race exists. The detection of races—a major problem in parallel debugging—is complicated by the presence of atomic critical sections. In programs without critical sections, the existence of a race is usually a bug leading to nondeterministic behavior. In programs with critical sections, however, accesses in parallel critical sections are not considered bugs, as the programmer, in specifying the critical sections, presumably intends them to run in parallel. Thus, a race detector should find "data races"—races between accesses *not* contained in atomic critical sections.

We present algorithms for detecting data races in programs written in the Cilk multithreaded language. These algorithms do not verify programs, but rather find data races in all schedulings of the computation generated when a program executes serially on a given input. We present two algorithms for programs in which atomicity is specified using locks, and an algorithm for programs using a proposed "guard statement" language construct to specify atomicity at a higher level than locks. We also extend each algorithm to handle critical sections containing parallelism.

We present the following algorithms, each characterized roughly in terms of the factor by which it slows down the computation being checked. In each case, memory usage is increased by roughly the same factor, unless otherwise stated.

- ALL-SETS. This algorithm checks programs with locks, with a slowdown factor of kL, where k is the maximum number of locks held simultaneously, and L is the maximum number of combinations of locks held during accesses to any particular location.
- BRELLY. This algorithm checks programs with locks, with a slowdown factor of only k. The gain in efficiency comes at the cost of flexibility and precision, since BRELLY detects violations of a proposed locking discipline that precludes some race-free locking protocols as well as data races.
- REVIEW-GUARDS. This algorithm checks programs with guard statements, with a slowdown factor of $\lg k$, where k is the maximum number of simultaneously guarded memory blocks. Space usage is increased by a constant factor.

The extensions of ALL-SETS and BRELLY that handle critical sections containing parallelism run a factor of k slower than the originals. The extension of REVIEW-GUARDS achieves the same performance as the original.

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

Acknowledgments

Charles Leiserson is the very best of advisors, and I am grateful to him for his generous, patient, and wise guidance, always given in good humor. His sheer delight in the science and art of computation is inspiring. I also thank him for letting me play with his toys (especially the quacking duck) while discussing technical matters in his office.

I am indebted to those with whom I have collaborated in much of the work presented in this thesis: Mingdong Feng, Charles Leiserson, Keith Randall, and Andy Stark. Each one has often impressed me with his diligence and acuity, and I count myself fortunate to have worked closely with them.

The Supercomputing Technologies Group at the MIT Lab for Computer Science (a.k.a. the Cilk group) has been a wonderfully stimulating and friendly environment. I thank the group members, both past and present, for their many tips and helpful and enjoyable discussions. Many thanks to Aske Plaat, Matteo Frigo, Mingdong Feng, Don Dailey, Phil Lisiecki, Andy Stark, Ching Law and, especially, Keith Randall—whose willingness to happily delve into any odd question, at most any time, still amazes me. I also thank Irena Sebeda for her administrative help, and for always stocking a good choice of refreshments at her desk.

I thank Anne Hunter of the EECS Department for her administrative assistance throughout my time at MIT.

I thank Victor Luchangco for providing me with invaluable feedback on several chapters of this thesis. I also want to explicitly thank Charles Leiserson for solicitously reading draft after draft of my thesis.

To Alice Wang, Lawrence Chang, and Nate Kushman: thanks for your company during many late nights chasing the thesis deadline together.

A special thank you to Nora Chen, who with a thoughtful remark during a casual lunch in Killian Court helped me decide to stay at MIT for a Master of Engineering degree.

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. The Xolas UltraSPARC SMP cluster, on which empirical work in thesis was done, was kindly donated by Sun Microsystems.

Finally, I thank my mother for her loving affection, steadfast support, and prayers. My debt to her, and to my late father, is inexpressible.

My naughty code has fallen sick
While racing on location l.
It's rather rude and nonatomic—
A nondeterministic hell.

Algorithms find the bug:
"Write to l, line 97."
Kiss the lemmas, give proofs a hug!
Praise deterministic heaven!

Contents

List of Figures

Chapter 1

Introduction: Race detection and atomicity

When two parallel threads access the same shared memory location, and at least one of them performs a write, a "determinacy race" exists: depending on how the two threads are scheduled, the accesses may occur in either order, possibly leading to nondeterministic program behavior, which is usually a bug. Race bugs are common to parallel programming, and they are notoriously hard to eliminate, since the nondeterministic runtime effects of a race are hard to identify—and even harder to duplicate—through informal runtime testing. Debugging tools that help detect races in parallel programs are therefore essential elements in any robust development environment for multithreaded programming.

To be as broadly useful as possible, race detection tools must handle programs that contain "atomicity," a common feature of multithreaded programs in a shared memory environment. Atomicity is the execution of parallel sections of code in a mutually exclusive fashion: the sections of code, called "critical sections," execute in any order but never interleave or operate simultaneously. The need for atomicity arises when a programmer intends that operations on the same memory locations be able to execute in parallel, in any order, as long as they do not interfere with each other by overwriting each other's intermediate or final results. For example, if a global counter is incremented in parallel by multiple threads and not printed out until after all the threads have finished, the printed value of the counter will be the same as long as the individual increment operations are atomic with respect to each other.

The presence of atomic operations in programs complicates the problem of race detection, since a determinacy race between mutually exclusive accesses should not be considered a potential bug. The programmer, in specifying that the operations be atomic, presumably intends that they run in parallel—atomicity has no meaning otherwise. To properly deal with atomicity, a race detection tool should only report

Parts of this chapter are based on "Detecting data races in Cilk programs that use locks," a paper by the present author, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark [3].

"data races," which are determinacy races between *nonatomic* accesses. To eliminate a data race, a user might either prevent the accesses from running in parallel, or make them mutually exclusive.

This thesis presents algorithms for detecting data races in the computations of Cilk programs containing atomicity. Cilk [4] is a multithreaded programming language based on C being developed at MIT's Lab for Computer Science. Linguistically, it adds to C [14] a "spawn" command, which creates subprocedures that execute in parallel, and a "sync" command, which forces a processor to wait for all spawned subprocedures to complete. We consider programs that use either of two mechanisms for specifying atomicity in Cilk: mutual-exclusion locks, which are available in version 5.1 of Cilk; and "guard statements," a new Cilk language construct proposed in this thesis. Locks are global variables which can be "held" by at most one processor at a time. Processors "lock" (or "acquire") and "unlock" (or "release") these lock variables in a mutually exclusive fashion: only one processor can hold a given lock at a time; any other processors trying to acquire a lock already held by another processor must wait until that processor releases the lock. Thus, atomicity can be provided by acquiring and releasing locks before and after critical sections. Figure 1-1 illustrates a data race in a Cilk program with locks.

The guard statement we propose provides atomicity at a higher level than locks, by allowing a user to directly specify which memory locations should be "guarded"—i.e. accessed atomically—within critical sections of code delimited by the usual C braces. Figure 1-2 illustrates guard statements, as well as data races in the context of guard statements.

The following algorithms are presented in this thesis, with worst-case performance bounds given for a Cilk computation that runs serially in time T, uses V shared memory locations, and either holds at most k locks simultaneously or guards at most k memory blocks simultaneously.

ALL-SETS. This algorithm detects data races in Cilk programs which use locks for atomicity. It runs in $O(LT(k + \alpha(V, V)))$ time and $O(kLV)$ space, where L is the maximum number of combinations of locks held during accesses to any particular location, and α is Tarjan's functional inverse of Ackermann's function, an extremely slowly growing function that is for all practical purposes at most 4. (Throughout this thesis, we use α to denote Tarjan's functional inverse of Ackermann's function.) ALL-SETS is the fastest known data-race detection algorithm, and seems to be practical when L is a small constant, which is the case for programs in which a particular data structure is always accessed using a particular lock or set of locks. There are programs, however, for which L grows with input size; for these programs ALL-SETS is less attractive and may be impractical. (For an example of such a program, see our discussion of a parallel maximum-flow code in Chapter 5.)

BRELLY. This algorithm is meant for Cilk programs which use locks for atomicity and for which the performance of ALL-SETS is inadequate. It detects violations of the "umbrella locking discipline," defined in this thesis, which precludes data

```
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 1-1: A Cilk program using locks, with a data race. The function `Cilk_lock()` acquires a specified lock, and `Cilk_unlock()` releases a currently held lock. The procedures `foo1`, `foo2`, and `foo3` run in parallel, resulting in parallel accesses to the shared variable `x`. The accesses in `foo1` and `foo2` are protected by lock `A` and hence do not form a data race. Likewise, the accesses in `foo1` and `foo3` are protected by lock `B`. The accesses in `foo2` and `foo3` are not protected by a common lock, however, and therefore form a data race. If all accesses had been protected by the same lock, only the value 3 would be printed, no matter how the computation is scheduled. Because of the data race, however, the value of `x` printed by `main` might be 2, 3, or 6, depending on scheduling, since the `x -= 3` and `x++` statements in `foo2` and `foo3` are composed of multiple machine instructions that may interleave, possibly resulting in a lost update to `x`.

races and also some complex, race-free locking protocols. Specifically, the umbrella discipline requires that within every parallel subcomputation, each shared memory location is protected by a unique lock. 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. One feature of the umbrella discipline is that it allows separate program modules to be composed in series without global agreement on a 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 particular 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.

The adoption of the umbrella discipline makes data-race detection easier, allowing Brelly to run in $O(kT\,\alpha(V,V))$ time and $O(kV)$ space—roughly a factor of L better than All-Sets. Since programs do not generally hold many

```
int x, y;

cilk void foo1() {
  guard(x; y) {
    x += 5;
    y += 5;
  }
}

cilk void foo2() {
  guard(x) {
    x -= 3;
    y -= 3;
  }
}
```

```
cilk void foo3() {
  guard(y) {
    x++;
    y++;
  }
}

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

Figure 1-2: A Cilk program using guard statements, with two data races. Each guard statement specifies which shared variables to guard—e.g. `guard(x, y)` guards `x` and `y`—and a section of code during which the variables should be guarded—e.g. { `x += 5; y += 5` }. Parallel accesses which access a shared location while it is guarded are atomic, so the accesses to `x` in `foo1` and `foo2`, and the accesses to `y` in `foo1` and `foo3`, do not form data races. The accesses to `x` and `y` in `foo2` and `foo3` do form data races against each other, however, since `foo2` does not guard `y` and `foo3` does not guard `x`. Because of these data races, the program may print any of 2, 3, or 6 for both `x` and `y`, as in the program in Figure 1-1.

locks simultaneously—k is almost always a small constant—these bounds are nearly linear.

If a program contains many nonrace violations of the umbrella discipline, debugging with Brelly may be impractical, since reports of data races may be buried under a deluge of reports of the nonrace violations. There exist, however, useful conservative heuristics that help Brelly determine whether a violation is indeed caused by a data race; in some cases, these heuristics drastically reduce the number of nonrace violations reported.

Review-Guards. This algorithm detects data races in Cilk programs that use guard statements for atomicity. It runs in $O(T(\lg k + \alpha(V, V)))$ time and $O(V + k)$ space. We know of no previous data-race detection algorithms explicitly designed for language constructs similar to the guard statement we propose.

The -Shared extensions. The basic versions of All-Sets, Brelly, and Review-Guards assume that critical regions do not contain parallelism. If critical sections do not contain parallelism, all accesses within a single critical section are never in parallel and so cannot be involved in races with each other. If critical sections do contain parallelism—as is necessary in some applications—the algorithms fail to detect data races between parallel accesses in the same critical section. Fortunately, the algorithms can be extended to correctly check critical sections containing parallelism. The extended versions of All-Sets and Brelly, called All-Sets-Shared and Brelly-Shared, run a factor of k slower and use a factor of k more space than the original versions. The extended version for guard statements, called Review-Guards-Shared, has the same asymptotic time and space bounds as Review-Guards. We know of no previous data-race detection algorithms which allow for critical sections containing parallelism.

These algorithms are not source-based verifiers; they do not ensure the detection of all possible data races in all possible schedulings of a program run on all possible inputs. Instead they check for data races that exist in all possible schedulings of the computation generated by the *serial execution* of a multithreaded program *on a given input*. Furthermore, like most race detectors, the algorithms attempt to find, in the terminology of Netzer and Miller [21], "apparent" data races—those that appear to occur in a computation according to the parallel control constructs—rather than "feasible" data races—those that can actually occur during program execution. The distinction arises because operations in critical sections may affect program control depending on the way threads are scheduled. An apparent data race between two threads in a given computation might be infeasible, because the computation itself may change if the threads are scheduled in a different order. Since the problem of finding feasible data races exactly is intractable [20], attention has naturally focused on the easier (but still difficult) problem of finding apparent data races.

Programs whose critical sections produce the same results independent of their execution order—i.e., programs with commutative critical sections—always produce, when running on a given input, a single computation in which an apparent race exists if and only if a feasible race exists [3]. On such "abelian" programs, the algorithms in this thesis can be used to guarantee that a program always produces the same behavior on a given input, regardless of scheduling. On all programs, these algorithms can be used to help find races and ensure correct, deterministic behavior in parallel codes that contain atomicity.

Related work

Since race detection is a major concern in parallel debugging, and locking is a common form of providing atomicity, automatic data-race detection in programs with locks has been studied extensively. Static race detectors [18] can sometimes determine whether a program will ever produce a data race when run on all possible inputs. Checking every possible control-flow of an arbitrary program is intractable, however, so 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 [8, 12, 16, 19], while others perform an on-the-fly analysis during program execution. On-the-fly debuggers directly instrument memory accesses via the compiler [6, 7, 9, 10, 15, 22], by binary rewriting [25], or by augmenting the machine's cache coherence protocol [17, 23]. The algorithms presented in this thesis detect data races dynamically, and can be used to create either post-mortem or on-the-fly debugging tools. (For convenience, we will describe them as on-the-fly algorithms.)

In previous work, Dinning and Schonberg's "lock-covers" algorithm [7] also detects all data races in a computation. Our 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 [23] 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.

Savage et al. [25] 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 data 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 understand the series/parallel relationships between threads, it does not fully 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 data races.

In comparison, our Brelly algorithm performs nearly as efficiently, is guaranteed to find all violations in a computation, and, importantly, supports a more flexible discipline. The umbrella discipline allows separate program modules to be composed in series without global agreement on a lock for each location, as seen above in the imagined program with three phases, each of which can use locks independently of the others.

Organization of this thesis

Chapter 2 provides a background on Cilk computations and SP-bags, the determinacy-race detection algorithm upon which the algorithms in this thesis are based. It also discusses locks in Cilk.

In Chapter 3, we present the All-Sets algorithm for detecting data races in computations with locks. We also give a general model for computations with critical sections containing parallelism, and present the -Shared extension of All-Sets for correctly handling such computations.

In Chapter 4, we consider the umbrella locking discipline and algorithms for detecting violations of the discipline. After showing that All-Sets may be impractical for some programs, we define the discipline itself. We then present the Brelly algorithm for detecting violations of the discipline and its -Shared extension for handling critical sections that contain parallelism. Finally, we describe several heuristics for conservatively determining whether a violation of the discipline is caused by a data race; these heuristics may increase Brelly's usefulness in practice.

Chapter 5 presents an empirical comparison of the runtime performance of All-Sets and Brelly on several Cilk programs and input. While these results are highly preliminary, we see that Brelly is, as predicted by our asymptotic bounds, faster than All-Sets for computations which hold many different sets of locks during accesses to particular locations.

In Chapter 6, we turn to the the problem of detecting data races in computations that use the proposed guard statement to specify atomicity. We first discuss the syntax, semantics, and possible implementations of the guard statement. Then, after showing how the algorithms for locks presented in Chapters 3 and 4 can be modified for guard statements but are not optimal, we present the efficient Review-Guards algorithm its -Shared extension for handling critical sections containing parallelism.

Chapter 7 summarizes the thesis, outlines further questions that arise from our work, and explores the dilemma of trying to debug parallel programs with algorithms that find apparent races instead of feasible ones.

Chapter 2

Background: Cilk and the Nondeterminator

This chapter provides background for the rest of the thesis. Because of Cilk's simple model of parallelism, computations of Cilk programs can be modeled as "series-parallel parse trees," which cleanly express the serial/parallel relationships between threads. We presents this model, as well as two lemmas about the series/parallel relationships between threads that will be useful for proving the correctness of our algorithms. We then discuss the Nondeterminator, an efficient determinacy-race detection tool for Cilk programs. SP-BAGS, the algorithm used by the Nondeterminator, is the basis for the algorithms in this thesis. Finally, we discuss locks in Cilk and define some basic locking terms that will be used when we present the algorithms dealing with locks in Chapters 3 and 4. Guard statements, the other form the atomicity considered in this thesis, are discussed in Chapter 6.

Modeling Cilk computations as SP-trees

Cilk extends the C programming language with two parallel commands: `spawn` creates a parallel subprocedure and `sync` causes a procedure wait for any subprocedures it has spawned to complete. In addition, normal C functions can be turned into Cilk ***procedures*** with the `cilk` tag, enabling them to be spawned. Cilk code has normal C semantics when executed serially (i.e. on a single processor): spawns of Cilk procedures behave like calls to normal C functions, with a depth-first execution of function calls leading to a linear call stack.

A Cilk ***computation*** is the execution of a Cilk program on a given input, scheduled in a particular way. In this thesis, we will refer only to the computation of a program executing *serially* on a given input. A thread is a maximal sequence of instructions not containing any parallel command. Threads may contain normal C function calls, since all normal C code is executed serially, but not spawns of Cilk

---

This chapter's discussion of Cilk computations and the SP-BAGS algorithm are based on [9], which contains a fuller treatment of the material only summarized here. The discussion of locks in Cilk and the fake read lock is from [3].

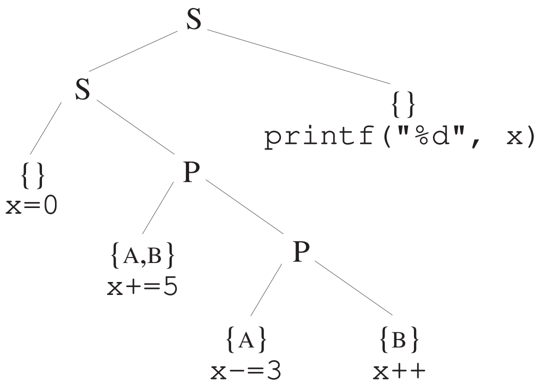


Figure 2-1: The series-parallel parse tree (SP-tree) for the Cilk program in Figure 1-1, abbreviated to show only the accesses to shared location `x`. Each leaf is labeled with a code fragment that accesses `x`, with the set of locks held during that access shown above the code fragment.

procedures, which execute in parallel. All procedure instances and threads in a computation have unique (numerical) IDs.

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

The computation dag generated by a Cilk program can itself be represented as a binary ***series-parallel parse tree***, as illustrated in Figure 2-1 for the program in Figure 1-1. 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. We use the term "logically" to mean with respect to the series-parallel control, not with respect to any additional synchronization through shared variables.

A parse tree allows the series/parallel relationship between two threads e_1 and e_2 to be determined by examining their least common ancestor, which we denote by $\textsc{lca}(e_1, e_2)$. If $\textsc{lca}(e_1, e_2)$ is a P-node, the two threads are logically in parallel, which we denote by $e_1 \parallel e_2$. If $\textsc{lca}(e_1, e_2)$ is an S-node, the two threads are logically in series, which we denote by $e_1 \prec e_2$, assuming that e_1 precedes e_2 in a left-to-right depth-first treewalk of the parse tree. The series relation $\prec$ is transitive.

It is sometimes possible to infer the series/parallel relationship between two threads based on the relation of the two threads to a common third thread. The following lemmas, proved in [9], show how to do so in two important cases. They will be used throughout the proofs of correctness for the algorithms in this thesis.

Lemma 2.1 Suppose that three threads e_1, e_2, and e_3 execute in order in a serial, depth-first execution of a Cilk program. If $e_1 \prec e_2$ and $e_1 \parallel e_3$, then $e_2 \parallel e_3$. ■

Lemma 2.2 (Pseudotransitivity of $\|$) Suppose that three threads e_1, e_2, and e_3 execute in order in a serial, depth-first execution of a Cilk program. If $e_1 \parallel e_2$ and $e_2 \parallel e_3$, then $e_1 \parallel e_3$. ∎

The Nondeterminator and the SP-BAGS algorithm

The race-detection algorithms in this thesis are based on the the SP-BAGS algorithm used in the Nondeterminator tool [9], which efficiently finds determinacy races (as opposed to data races) in Cilk programs. SP-BAGS is the fastest published determinacy-race detection algorithm that finds a determinacy race in a computation if and only if one exists. The algorithm serves as a strong foundation from which to attack the related, but more difficult problem of data-race detection.

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. While executing the program, SP-BAGS maintains an "SP-bags" data structure based on Tarjan's nearly linear-time least-common-ancestors algorithm [27]. The SP-bags data structure allows the algorithm to determine the series/parallel relationship between the currently executing thread and any previously executed thread in $O(\alpha(V,V))$ amortized time, where V is the size of shared memory. In addition, SP-BAGS maintains a "shadow space" where information about previous accesses to each location is kept. This information is used during an access to check previous threads that have accessed the same location for data races. Implementing the SP-BAGS algorithm involves modifying the Cilk compiler to instrument, according to logic of SP-BAGS, each memory access and parallel control statement For a Cilk program that runs in T time serially and references V shared memory locations, the SP-BAGS algorithm runs in $O(T\,\alpha(V,V))$ time and uses $O(V)$ space.

Each of the algorithms in this thesis uses the SP-bags data structure to determine the series/parallel relationship between threads, and, like SP-BAGS, executes a Cilk program serially on a given input, in left-to-right depth-first order, with appropriate race-detection logic being executed at each memory accesses and, in some case, at each parallel control statement. In addition to spawn and sync, parallel control statements include the return from a spawned Cilk procedure when it finishes.

Furthermore, like SP-BAGS, each of the algorithms in this thesis runs in a per-location manner. Some data structures are kept globally during the serial execution—e.g. the set of currently held locks—but much data is kept per-location, and independently across locations, in various shadow spaces of shared memory. Since our task is to detect data races, which occur on specific memory locations, it is useful to think of the algorithms as executing the same computation multiple times in sequence, with a different location being checked for data races at each instance. In reality, of course, each algorithm runs a computation just once, with per-location information being kept in the shadow spaces independently.

Locks in Cilk

Release 5.1 of Cilk [4] provides the user with mutual-exclusion locks, including the command `Cilk_lock` to acquire a specified lock and `Cilk_unlock` to release a currently held lock. Any number of locks may be held simultaneously, and locks can be acquired and released however the user likes. We do assume for simplicity, however, that a lock is never released by a spawned child of the procedure which acquired it and that no locks are ever held when a Cilk procedure returns. For a given lock `A`, the sequence of instructions from a `Cilk_lock(A)` to its corresponding `Cilk_unlock(A)` is called a ***critical section***, and we say that all accesses in the critical section are ***protected*** by lock `A`. If a critical section does not contain parallelism, we say it is a ***serial critical section***.

The ***lock set*** of an access is the set of locks held when the access occurs. The ***lock set*** of several accesses is the intersection of their respective lock sets. In programs with serial critical sections, a data race can be defined in terms of lock sets as follows: 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 ***data race*** exists.

To simplify the description and analysis of the race detection algorithms in Chapter 3umbrella, we will 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 it is never actually acquired and released (as it does not exist). The use of R-LOCK simplifies the description and analysis of our race detection algorithms, because it allows us to state the condition for a data race more succinctly: *if the lock set of two parallel accesses to the same location is empty, then a data race exists.* By this condition, a data race (correctly) does not exist for two read accesses, since their lock set contains the R-LOCK.

Chapter 3

Data-race detection in computations with locks

In this chapter, we consider the problem of precisely detecting data races in programs that use locks to provide atomicity, presenting the All-Sets algorithm and an extension of it that handles critical sections containing parallelism. All-Sets is precise: it detects a data race in a computation if and only if one exists. All-Sets is also reasonably efficient for many computations. On a Cilk program running serially in time T using V shared memory locations, All-Sets runs in $O(LT\,(k + \alpha(V,V)))$ time using $O(kLV)$ space, where k is the maximum number of locks held simultaneously and L is the maximum number of combinations of locks held during accesses to any particular location. For programs which always use the same lock or set of locks to access particular locations, L is a small constant and All-Sets will likely be practical. All-Sets-Shared, the extension of All-Sets for handling critical sections containing parallelism, runs only a factor of k slower and uses only a factor k more space than the original in the worst case.

This chapter is organized as follows. Section 3.1 presents the All-Sets algorithm, showing how it conceptually remembers every shared memory access during an execution, but with at most one shadow space entry per distinct lock set per location. Then, in Section 3.2, we extend the SP-tree model of Cilk computations (see Chapter 2) to provide for critical sections containing parallelism. This extended SP-tree model is the basis for the operation of All-Sets-Shared, presented in Section 3.3, as well as the -Shared version of the Brelly algorithm for detecting violations of the umbrella discipline, presented in Chapter 4.

3.1 The All-Sets algorithm

The All-Sets algorithm finds data races in Cilk computations that use locks, assuming critical sections do not contain parallelism. In this section, we see how the algorithm, while conceptually keeping track of every accesses to every location during

Section 3.1 is based on joint work published in [3].

an execution, actually prunes redundant entries from the shadow space to ensure that at most one entry per lock set per location is recorded. This logic leads directly to factor of L in the performance bounds of ALL-SETS: on a Cilk program running serially on given input in time T using V space, ALL-SETS runs in $O(LT(k + \alpha(V, V)))$ time using $O(kLV)$ space, where k is the maximum number of locks held simultaneously and L is the maximum number of combinations of locks held during accesses to any particular location. We prove the correctness of ALL-SETS and show these bounds at the end of this section.

Like the efficient SP-BAGS algorithm used by the original Nondeterminator (Chapter 2), upon which it is based, ALL-SETS executes a Cilk program on a particular input in serial depth-first order. The ALL-SETS algorithm also uses the SP-bags data structure from SP-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 entry *lockers*$[l]$ stores a list of ***lockers***: threads that access location l, each paired with the lock set that is held during the access. If $\langle e, H \rangle \in$ *lockers*$[l]$, then 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_lock(C);
WRITE(l)
Cilk_unlock(C);
Cilk_unlock(A);
```

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

The ALL-SETS algorithm is shown in Figure 3-1. Intuitively, this algorithm records all lockers, but it is careful to prune redundant lockers, keeping at most one locker per distinct lock set per location. Lines 1–3 check to see if a data race has occurred and report any violations. Lines 4–11 then add the current locker to the *lockers* shadow space and prune redundant lockers. While it is only necessary to prune lockers with identical lock sets to achieve the stated performance bounds, ALL-SETS is also able, in some cases, to prune a locker if its lock set is a proper subset of another locker's lock set.

When a location l is accessed outside any critical section, the lock set H contains either the empty set or the singleton R-LOCK set, depending on whether the access is a write or a read, respectively. Thus, the original SP-BAGS algorithm, which finds determinacy races in Cilk programs without locks, can be considered a special case of ALL-SETS in which the only two lock sets that appear in the *lockers* shadow space are the empty set and the singleton R-LOCK set.

```
ACCESS(l) in thread e with lock set H
1  for each ⟨e', H'⟩ ∈ lockers[l]
2      do if e' ∥ e and H' ∩ H = {}
3            then declare a data race
4  redundant ← FALSE
5  for each ⟨e', H'⟩ ∈ lockers[l]
6      do if e' ≺ e and H' ⊇ H
7            then lockers[l] ← lockers[l] − {⟨e', H'⟩}
8         if e' ∥ e and H' ⊆ H
9            then redundant ← TRUE
10 if redundant = FALSE
11    then lockers[l] ← lockers[l] ∪ {⟨e, H⟩}
```

Figure 3-1: The ALL-SETS algorithm. The operations for the `spawn`, `sync`, and `return` actions that maintain the SP-bags data structure are unchanged from the SP-BAGS algorithm on which ALL-SETS is based. Additionally, the `Cilk_lock` and `Cilk_unlock` functions must be instrumented to appropriately add and remove locks from H, the set of currently held locks.

We now prove that ALL-SETS correctly finds data races.

Theorem 3.1 Consider a Cilk program with locks and serial critical sections. The ALL-SETS algorithm detects a data race in the computation of this program running serially on a given input if and only if a data race exists in the computation.

Proof: ($\Rightarrow$) 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 data race.

($\Leftarrow$) Assuming a data race exists in a computation, we shall show that a data race is reported. If a data 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 which has a data 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 $H_1 \cap H_2 = \{\}$ by definition of a data race.

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 of $\parallel$ (Lemma 2.2), 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 2.1, 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. ∎

We now show the performance bounds for ALL-SETS.

Theorem 3.2 Consider a Cilk program—one with locks and serial critical sections—that, on a given input, executes serially in time T, references V shared memory locations, and holds at most k locks simultaneously. The ALL-SETS algorithm checks this computation for data races in $O(LT(k+\alpha(V,V)))$ 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, for some location l, no two lockers in *lockers*$[l]$ for have the same lock set, because the logic in lines 4–11 ensures 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 $O(k)$ space, so the space needed for *lockers* is $O(kLV)$. The length of the list *lockers*$[l]$ at the time of an access determines the number of series/parallel relationships that are tested during that access. In the worst case, we need to perform $2L$ such tests and $2L$ set operations per access (line 2 and lines 6 and 8). Each series/parallel test takes amortized $O(\alpha(V,V))$ time, and each set operation takes $O(k)$ time (lock sets can be stored in sorted order). Therefore, the ALL-SETS algorithm runs in $O(LT(k + \alpha(V,V)))$ time, since T is an upper bound on the number of accesses. ∎

The bounds proven in Theorem 3.2 show that the performance of ALL-SETS is directly proportional to the value of L for a program running on a particular input. If a program uses many distinct lock sets to access each memory location, and especially if the number locks sets to access individual locations grows with input size, the performance of ALL-SETS may be inadequate, as we will see in Chapter 5. For many programs, however, both L and k are small constants—e.g. if each data structure is always accessed with a single lock or lock set—and so the bounds for ALL-SETS are good. In these cases, empirical performance is also good, as we also see in Chapter 5.

3.2 Locked critical sections containing parallelism

In this section, we extend the SP-tree model of Cilk computations to provide for programs with locked critical sections containing parallelism. This extended model is the basis of the detection algorithms which correctly handle critical sections containing parallelism: ALL-SETS-SHARED in Section 3.3 and BRELLY-SHARED in Section 4.3. (Some of the concepts introduced here are borrowed in Section 6.4, which presents the -SHARED extension of the REVIEW-GUARDS algorithm for detecting data races

```
int x, y;
Cilk_lockvar A, B;

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

```
cilk int main() {
  Cilk_lock_init(A);
  Cilk_lock_init(B);
  x = y = 0;
  Cilk_lock(A);
  spawn increment();
  spawn increment();
  sync;
  Cilk_unlock(A);
  printf("%d %d", x, y);
}
```

Figure 3-2: A Cilk program with a critical section containing parallelism. The critical section protected by lock `A` in `main` spawns two instances of `increment`, the accesses in which are not protected against each other by lock `A`.

in programs using "guard statements" rather than locks for atomicity.) After seeing why our definition of data races based on the simple SP-tree model is faulty when critical sections contain parallelism, we explain the model itself, and finally show how to use it to redefine the notion of a data race to provide for programs with critical sections containing parallelism.

Under the assumption that critical sections contain only serial code, two parallel accesses are mutually exclusive, and therefore not in a data race, if they hold a common lock, since the accesses, being in parallel, are protected by separate "instances" of the lock—i.e. the common lock is held during the accesses due to separate `Cilk_lock` statements in the computation. The ALL-SETS algorithm, which assumes no parallelism within critical sections, takes advantage of this fact in its basic logic: it never declares a data races between accesses that hold a lock in common. If critical sections can contain parallelism, however, this logic is faulty, since two parallel accesses within the same locked critical section may hold the same instance of a common lock—i.e. they ***share*** the same instance of that lock—and so the lock does not protect the accesses against each other. Indeed, any parallel accesses within the same critical section share the instance of the lock that protects the critical section.

For example, consider the Cilk program in Figure 3-2. This program acquires lock `A`, spawns two instances of the `increment` procedure, syncs, and then releases lock `A`. The accesses to the shared variable `x` in the two instances of `increment` form a data race, even though lock `A` is held during both accesses. Lock `A` does not protect the accesses against each other, since they share the same instance of `A`—the one acquired in `main`. For the same reason, lock `A` does not protect the accesses to `y` in `increment` against each other, but no data race exists between these accesses since they are protected against each other by lock `B`, which is acquired separately by each instance of `increment`.

How can we formally model critical sections that contain parallelism and specify data races in this model? We first need to specify what parallelism is allowed within critical sections. Intuitively, we would like the easy-to-understand serial semantics of any critical section to hold when run in parallel: if a lock is held during some access in the serial execution, it should be held during that access in any parallel execution. Fortunately, ensuring these semantics requires only one rule to be followed: all parallel procedures spawned within a critical section (i.e. those spawned while a lock is held) must complete before the end of the critical section. Linguistically, this means there must be a sync between the final spawn and the `Cilk_unlock` statement in the execution of a critical section, if the section contains any spawns at all. Without this sync after the last spawn in a critical section, the `Cilk_unlock` statement that ends the section would be logically in parallel with at least one of the spawns in the critical section (perhaps all of them), meaning that the lock might be released before these spawns finish and therefore would not protect them against other parallel threads holding the same lock. With the sync, however, all the spawned procedures (together with all the serial operations) in the critical section are guaranteed to be executed while the section's lock is held. Thus, we require a sync at the end of critical sections.[1]

There is no analogous need for a spawn to be followed by a sync before the next `Cilk_lock` statement, even though a spawn followed by a `Cilk_lock` without an intervening sync would result in the spawned procedure being logically in parallel with the lock acquisition. Since the `Cilk_lock` may be scheduled after the time the spawned procedure runs, we must assume that the spawned procedure is not protected by the lock, which, as desired, corresponds to the intuitive serial semantics of the code.

To model a computation with critical sections containing parallelism using SP-trees, we define a ***locked P-node*** to be a P-node corresponding to either the first spawn in a critical section or the first spawn after a sync within a critical section, which is immediately preceded, in the depth-first left-to-right treewalk, by the acquisition of one or more locks and immediately succeeded by the release of these locks. These locks are said to be ***held across*** the P-node, as are any locks acquired at any time before and released at any time after the depth-first traversal of a P-node's subtree. Any parallel accesses in an SP-tree that are descendents of a particular locked P-node are not protected against each other by any lock held across the P-node, since they share the same instance of the lock. Figure 3-3 shows the SP-tree, a with a locked P-node, for the program in Figure 3-2.

Is this notion of locked P-nodes sufficient to model all critical sections containing parallelism? In general, any critical section in a Cilk program can be modeled at runtime as a sequence of zero or more "sync blocks" followed by a final thread;

[1] In fact, we suggest that the semantics of `Cilk_unlock` be extended to include an implicit sync. The extra sync would cause no overhead when no spawns are outstanding at the execution of a `Cilk_unlock` statement, since syncs in serial code are ignored by the Cilk compiler. If `Cilk_unlock` does not include an implicit sync, then violations of the sync-before-unlock rule can be easily detected during a depth-first serial execution: keep a global flag that is set at each sync and `Cilk_lock` statement, cleared at each spawn, and verified to be to set at each `Cilk_unlock` statement.

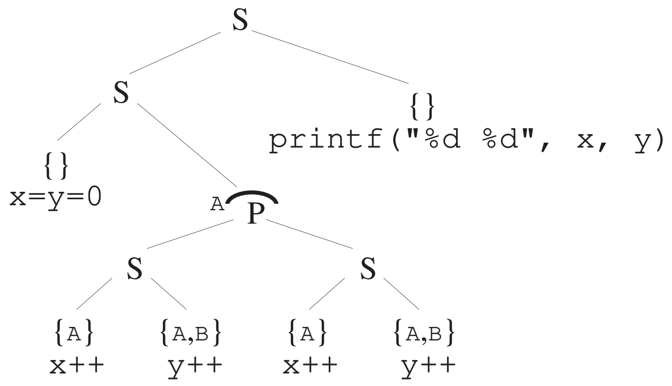


Figure 3-3: The SP-tree for the Cilk program in Figure 3-2. Each leaf is labeled a code fragment that accesses either `x` or `y`, the two shared memory variables in the program, with the lock set for that access shown above the code fragment. The P-node representing the first spawn of `increment` (the only P-node in the tree) is locked by `A`, indicated by the labeled arc over it. This arc represents the fact that a single instance of lock `A` is held across all the accesses in the subtree rooted by the P-node, so that these accesses are not protected against each other by `A`, despite the fact that `A` is a member of each of their lock sets.

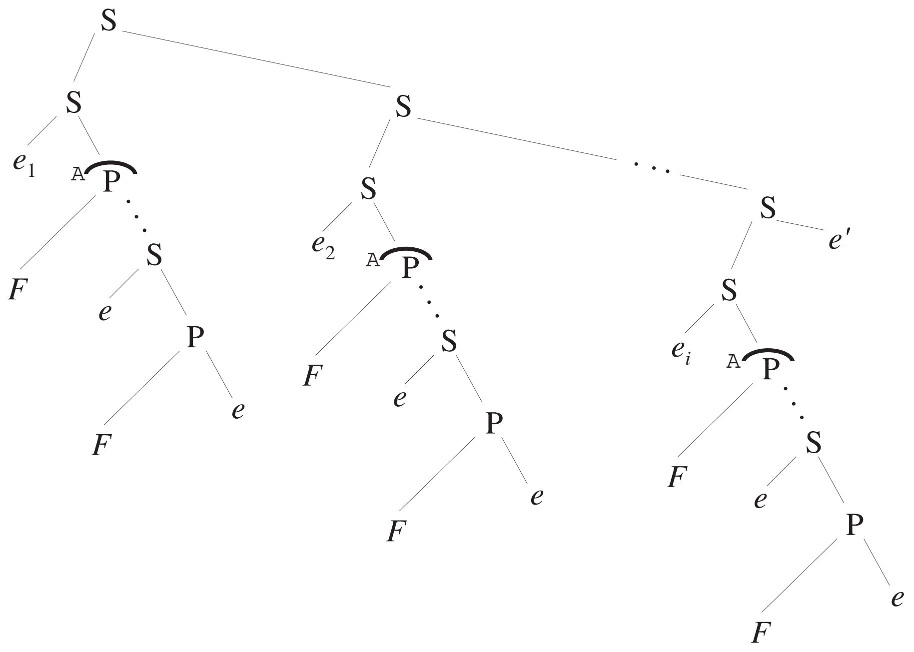


Figure 3-4: The SP-tree, with locked P-nodes, representing the generalized form of a critical section protected by lock `A`. Although the lock is held across this entire tree, the P-nodes corresponding to the first spawns in each sync block are shown as locked P-nodes, as if separate instance of `A` were held across them. Also, the first threads of each sync block, $e_1, e_2, \ldots, e_i$, and the thread e' at the end of the critical section— none of which are descendents of a locked P-node—are considered to hold their own instances of `A` (not shown).

each sync block consists of a one or more spawns interleaved with serial code and is terminated by a sync. In other words, a critical section during execution has the form

$\texttt{Cilk\_lock(A);}$
e_1; spawn F; e; spawn F; e; $\ldots$; spawn F; e; sync;
e_2; spawn F; e; spawn F; e; $\ldots$; spawn F; e; sync;
$\vdots$
e_i; spawn F; e; spawn F; e; $\ldots$; spawn F; e; sync;
e';
$\texttt{Cilk\_unlock(A);}$

where each of $e_1, e_2, \ldots, e_i$ is the thread at the beginning of a sync block, each e is a (possibly empty) thread, each F is a Cilk procedure, and e' is the final (possibly empty) thread after the last sync block before the end of the critical section. (Note that the threads in a critical section may acquire or release other locks, as the critical section may contain or overlap other critical sections.) Because a single critical section may contain several sync blocks, it might appear that locked P-nodes, which are explicitly defined to correspond to locks held across only a single sync block, are insufficient to model all critical sections with parallelism. Locked P-nodes are sufficient, however, since a series of sync blocks are, by definition, logically in series, and therefore contain no races among them. From the point of view of race detection, a series of several sync blocks all surrounded by a single lock/unlock pair is equivalent to a series of the same sync blocks, each individually surrounded by the same lock/unlock pair. Furthermore, it is also equivalent to consider the threads $e_1, e_2, \ldots, e_i$ at the beginnings of sync blocks, and the thread e' at the end of a critical section, to be protected by their own instances of the same lock, since they too are logically in series with the rest of the sync blocks. Figure 3-4 shows the SP-tree, with locked P-nodes, for the generalized form of a critical section shown above. Notice that the threads $e_1, e_2, \ldots, e_i$ and e', and the sync blocks themselves, are all logically in series with each other, and that any parallel threads within the critical section are descendents of the same locked P-node.

The problem of detecting data races in computations with critical sections that may contain parallelism reduces, then, to the problem of finding data races in an SP-tree which may include locked P-nodes. But what comprises a data race in such an SP-tree? Before answering this question, it will be useful to define several further notions. The ***lock-sharing depth*** of any node in an SP-tree (internal or leaf) is the number of locked P-nodes among its ancestors, including itself. The ***depth-i lock set*** of an access is the set of locks held during the access minus the locks which are held across the access's ancestor P-nodes with lock-sharing depth less than or equal to i. Note that the depth-0 lock set for any access is the set of all currently held locks. See Figure 3-5 for an illustration of lock-sharing depth and depth-based lock sets.

A precise formulation of a data race in an SP-tree with locked P-nodes can be given based on depth-based lock sets and the following notion of depth-based cousinhood: if the the lock-sharing depth of the least-common ancestor of two accesses in an SP-tree is i, the accesses are ***depth-i cousins***. Furthermore, if the depth-i lock sets of the accesses have no locks in common, then they form a data race.

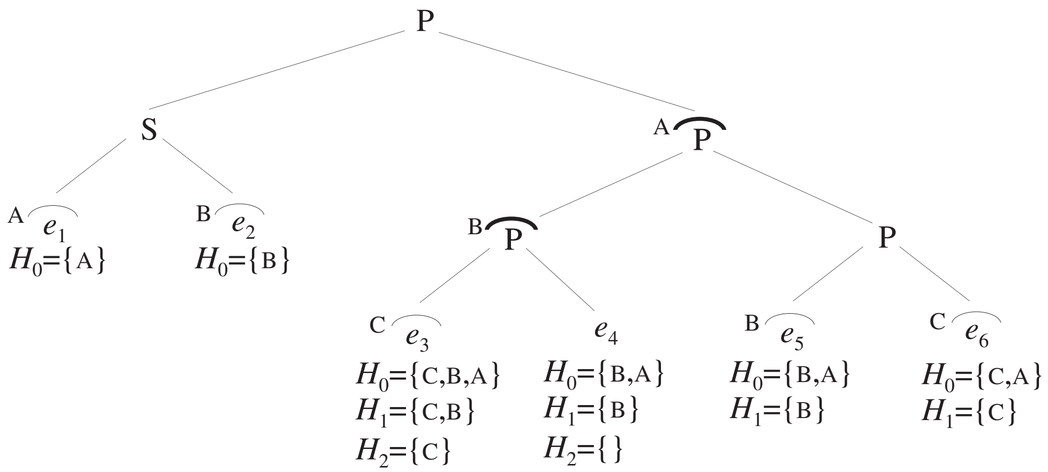


Figure 3-5: An SP-tree containing locked P-nodes. Each leaf represents a thread that accesses some shared location l. Locked P-nodes are marked with bold arcs above them, annotated with the lock acquired just before and after the traversal of their subtrees. For ease of identifying where any particular lock is acquired and released, locks acquired at the beginning and released at the end of a single thread, and thus protecting a serial critical section, are shown in a similar way: a thin arc, annotated with the lock, appears above thread, as with e_1. Under each thread are the depth-based lock sets of the thread's memory access, with H_0 being the depth-0 lock set, H_1 being the depth-1 lock set, and so on, up the the lock-sharing depth of the access. For example, the access in e_4 is at lock-sharing depth 2, since there are two locked P-nodes among its ancestors, the one at depth 1 locked by A and the one at depth 2 locked by B. For this access, the depth-0 lock sets includes all locks held ($\{A, B\}$), while these locks are successively subtracted, first A at depth 1 and then B at depth 2, to form the depth-1 and depth-2 lock sets. The lock sets for the access in e_3, which is also at lock-sharing depth 2, are the same as those for the access in e_4, except that lock C is additionally included in every lock set, since C does not lock a P-node at all, but rather protects a serial critical section in e_3

For example, consider Figure 3-5. The parallel accesses in e_1 and e_3 are depth-0 cousins, since their least-common ancestor is the P-node at the root of the entire tree, which is not locked. The intersection of their depth-0 lock sets contains lock A, which indeed protects them against each other, since e_1 is protected by its own instance of A and e_3 by the instance of A that locks the right child of the root. The access in e_1 is likewise a depth-0 cousin of the accesses in e_4, e_5, and e_6, and is also protected against them by lock A. The same holds for the relationship between the access in e_2 and the ones in e_3, e_4, and e_5, except that in these cases the accesses are protected against each other by lock B, different instance of which are held across e_3 and e_4 (the one locking LCA(e_3, e_4)) and e_5 (its own instance). No lock protects the accesses in e_2 and e_6 from each other, indicated by the empty intersection of their depth-0 lock sets.

Now, consider the depth-1 cousins in the tree. The access in e_4 is a depth-1 cousins of the accesses in e_5 and e_6, since their least-common ancestor is the right child of the root, and there is one locked P-node (the ancestor itself) along the path from that ancestor to the root. The access in e_5 is protected against the one in e_4 by B (which is in the respective depth-1 lock sets), since, as we saw above, they are performed under different instances of the lock. No lock protects the accesses in e_4 and e_6 against each other, and the intersection of their depth-1 lock sets is accordingly empty. The accesses in e_5 and e_6 are also depth-1 cousins that are unprotected against each other, as seen by their depth-1 lock sets.

Finally, consider the two depth-2 cousins in the tree: e_3 and e_4. The accesses in these threads, despite both being performed while both locks A and B are held, are not protected against each other, since they hold no unshared lock in common, as indicated by the empty intersection of their depth-2 lock sets.

The following theorem and corollary show that our formulation of data races based on depth-based cousinhood and lock sets is correct.

Theorem 3.3 Suppose e_1 and e_2 are depth-i cousins that access a shared location in parallel, and let h be a lock that is held by both accesses. The accesses share the same instance of h if and only if h is not in in the intersection of their depth-i lock sets.

Proof: ($\Rightarrow$) Suppose the accesses share the same instance of h. Then they must both be descendents some P-node p_h locked by h. Let p be the deepest locked P-node of which both e_1 and e_2 are descendents. By the definition of cousinhood, the lock-sharing depth of p is i, and so, by definition of depth-based lock sets and the fact that p_h must be equal to or an ancestor of p, neither of the accesses' depth-i lock sets contains h.

($\Leftarrow$) Suppose h is not in the intersection of the accesses' depth-i lock sets. Assume without loss of generality that h is not in the depth-i lock set of e_1. By definition of depth-based lock sets, we know that h is held across some P-node p among e_1's ancestors of lock-sharing depth i or less. Since e_1 and e_2 are depth-i cousins, e_2 is also a descendent of p. Thus, they share the same instance of h, namely, the one that is held across p. ∎

Corollary 3.4 Two parallel accesses that are depth-i cousins form a data race if and only if the intersection of their depth-i lock sets is empty. ∎

3.3 The All-Sets-Shared algorithm

Now that we have a model for Cilk computations with critical sections that contain parallelism, and further know how to recognize a data race in this model, we can extend All-Sets to handle such computations correctly. In this section we describe All-Sets-Shared, an extension of All-Sets based on this model. We show that it correctly detects data races in such computations, and prove that it runs a factor of k slower and uses a factor of k more space than the original All-Sets algorithm. The operation of All-Sets-Shared, which is conceptually based on the execution of multiple instances of the original All-Sets at various lock-sharing depths during each access, also provides a model for the operation of Brelly-Shared, discussed in Section 4.3.

The logic of the All-Sets-Shared algorithm is based on Corollary 3.4: at each shared memory access, for $i = 0, 1, \ldots, D$ where D is the lock-sharing depth of the access, compare the access's depth-i lock set with the depth-i lock sets of its previously executed depth-i cousins, declaring a race whenever the intersection of two lock sets is empty. To do this, All-Sets-Shared in essence runs one instance of All-Sets for each depth of cousinhood at each memory access, keeping a separate set of *lockers* for each depth. For an intuition of how this might work, again consider Figure 3-5, and imagine that All-Sets were run on the entire tree, using depth-0 lock sets; on the subtree rooted by the right child of the root, using depth-1 lock sets; and on the subtree rooted by $\text{LCA}(e_3, e_4)$, using depth-2 lock sets. Rather nice how all data races would be found correctly, isn't it?

The All-Sets-Shared algorithm maintains a global variable D, the lock-sharing depth of the current access, and global lock sets $H^{(i)}$ for $i = 0, 1, \ldots, D$, where $H^{(i)}$ is the current access's depth-i lock set. To help maintain D, the algorithm also tags each entry in the Cilk procedure call stack with a *sync-depth* field, which records the number of subprocedures associated with locked P-nodes, spawned from the procedure, that have yet to be synchronized. There is a dynamic global array *pstack*, indexed $pstack^{(1)}, pstack^{(2)}, \ldots, pstack^{(D)}$, containing a stack of the IDs of the locked P-nodes which are the ancestors of the current access, from oldest to most recent. (The ID of the Cilk procedure instance associated with a locked P-node serves nicely as the node's ID.)

For each shared memory location l, All-Sets-Shared keeps a list of depth-0 lockers in $lockers^{(0)}[l]$, a list of depth-1 lockers in $lockers^{(1)}[l]$, a list of depth-2 lockers in $lockers^{(2)}[l]$, and so on, up to the lock-sharing depth of the most-recent access to l, which is stored in *lockers-depth*$[l]$. Analogously to All-Sets, each ***locker*** in $lockers^{(i)}[l]$ is a pair $\langle e, H \rangle$, where e is the ID of a thread which accesses l at a lock-sharing depth greater than or equal to i, and H is the depth-i lock set held during the access. Finally, there is a P-node ID $pid[lockers^{(i)}[l]]$ associated with each $lockers^{(i)}[l]$

for $i = 1, 2, \ldots, \mathit{lockers\text{-}depth}[l]$, indicating the depth-$i$ locked P-node among the ancestors of the last access to l. (There is no need for a *pid* at depth 0 since a locked P-node is never at lock-sharing depth 0.)

The logic for All-Sets-Shared is shown in Figure 3-6, including the actions for locking and unlocking; spawns, syncs, and returns; and memory accesses. (As in all the algorithms for computations with locks, All-Sets-Shared uses the r-lock trick to avoid distinguishing between reads and writes.) The algorithm maintains D, the current lock-sharing depth, in lines 2–3 of Spawn, where D is incremented whenever a spawn occurs while a lock is held; and in lines 1–2 of Sync, where D is decremented by $\mathit{sync\text{-}depth}[F]$, which in essence is the number of locked P-nodes whose subtrees are being completed at the sync. The current depth-based lock sets, $H^{(0)}$ through $H^{(D)}$, are updated in Lock and Unlock, which simply add or remove a lock from each lock set; and in line 4 of Spawn, which adds a new empty lock set $H^{(D)}$ whenever D is incremented. The stack of locked P-nodes among the ancestors of the current access is maintained by line 5 of Spawn. (For simplicity, we do not show obsolete entries of $\mathit{pstack}^{(i)}$—those with index i greater than the current D—being cleared.)

With D and the depth-based lock sets properly maintained, the algorithm performs two phases during each access to a shared memory location l, shown in Access. First, in lines 1–4, it deletes the lists of lockers for the location whose *pid*'s are no longer in *pstack*, as these lockers are no longer relevant because the computation has reached a different part of the SP-tree (line 3)[2]; and it updates the *pid*'s of each of l's lists of lockers to be equal to the ID of the P-node stored in *pstack* at the same depth (line 4), so that future accesses can tell whether these lockers are relevant to them. To help with this first phase, *lockers-depth* is updated at each access in line 5.

The second phase of Access(l), in lines 6–17, executes the logic of All-Sets for each lock-sharing depth of the access (0 through D), checking the access against previously recorded lockers and updating the lists of lockers appropriately. Notice that lines 7–17 exactly duplicate the code of All-Sets (Figure 3-1), except that the lock sets and lockers considered at each iteration are for a specific lock-sharing depth.

Our proof of the correctness of All-Sets-Shared assumes that the global variables D and *pstack*, and the *pid*'s of the lists of lockers and *lockers-depth* for each location, are maintained correctly according to the following lemma, itself stated without proof.

Lemma 3.5 During an access to a location l in All-Sets-Shared, the access's lock-sharing depth is recorded in D, and the IDs of the locked P-nodes among the access's ancestors in the SP-tree are recorded, oldest to most recent, in $\mathit{pstack}^{(1)}$, $\mathit{pstack}^{(2)}$, …, $\mathit{pstack}^{(D)}$. Also, at the start of Access(l), the lock-sharing depth of the most-recently executed access to l is recorded in $\mathit{lockers\text{-}depth}[l]$, and the IDs of

[2]As shown in Figure 3-6, the algorithm deletes only those lists of lockers which are at or below the current depth D; any deeper lists of lockers are ignored because the iteration in line 6 only goes to D. If the execution ever reaches a deeper access, the left-over lists of lockers are deleted then. Of course, the lists of lockers deeper than D could be deleted up front at every access: it makes no difference whether they are deleted now or later.

SPAWN procedure F' from procedure F

if $H^{(D)} \neq \{\}$
  then $D \leftarrow D + 1$
    $\mathit{sync\text{-}depth}[F] \leftarrow \mathit{sync\text{-}depth}[F] + 1$
    $H^{(D)} \leftarrow \{\}$
    $\mathit{pstack}^{(D)} \leftarrow F'$
update SP-bags data structure according to the SPAWN logic in SP-BAGS

LOCK(a)

for $i \leftarrow 0$ **to** D
  do $H^{(i)} \leftarrow H^{(i)} \cup a$

UNLOCK(a)

for $i \leftarrow 0$ **to** D
  do $H^{(i)} \leftarrow H^{(i)} - a$

SYNC in procedure F

$D \leftarrow D - \mathit{sync\text{-}depth}[F]$
$\mathit{sync\text{-}depth}[F] \leftarrow 0$
update SP-bags data structure according to the SYNC logic in SP-BAGS

RETURN from procedure F' to procedure F

update SP-bags data structure according to the RETURN logic in SP-BAGS

ACCESS(l) in thread e

for $i \leftarrow 1$ **to** D
  do if $i > \mathit{lockers\text{-}depth}[l]$ or $\mathit{pstack}^{(i)} \neq \mathit{pid}[\mathit{lockers}^{(i)}[l]]$
    then $\mathit{lockers}^{(i)}[l] \leftarrow \{\}$
      $\mathit{pid}[\mathit{lockers}^{(i)}[l]] \leftarrow \mathit{pstack}^{(i)}$
$\mathit{lockers\text{-}depth}[l] \leftarrow D$
for $i \leftarrow 0$ **to** D
  do for each $\langle e', H' \rangle \in \mathit{lockers}^{(i)}[l]$
      do if $e' \parallel e$ and $H' \cap H^{(i)} = \{\}$
        then declare a data race
    $\mathit{redundant} \leftarrow$ FALSE
    for each $\langle e', H' \rangle \in \mathit{lockers}^{(i)}[l]$
      do if $e' \prec e$ and $H' \supseteq H^{(i)}$
          then $\mathit{lockers}^{(i)}[l] \leftarrow \mathit{lockers}^{(i)}[l] - \{\langle e', H' \rangle\}$
        if $e' \parallel e$ and $H' \subseteq H^{(i)}$
          then $\mathit{redundant} \leftarrow$ TRUE
    if $\mathit{redundant}$ = FALSE
      then $\mathit{lockers}^{(i)}[l] \leftarrow \mathit{lockers}^{(i)}[l] \cup \left\{\langle e, H^{(i)} \rangle\right\}$

Figure 3-6: The ALL-SETS-SHARED algorithm for detecting data races in Cilk computations with critical sections containing parallelism. LOCK and UNLOCK are executed when acquiring and releasing locks; SPAWN, SYNC, and RETURN at parallel control statements; and ACCESS when reading or writing a shared memory location.

the locked P-nodes among that last access's ancestors are recorded, oldest to most recent, in $pid[lockers^{(0)}[l]], pid[lockers^{(1)}[l]], \ldots, pid[lockers^{(lockers\text{-}depth[l])}[l]]$. ∎

We now show that ALL-SETS-SHARED correctly detects data races in Cilk computations with critical sections that may contain parallelism.[3]

Theorem 3.6 Consider a Cilk program with locks and critical sections containing perallelism, restricted as described in Section 3.2. The ALL-SETS-SHARED algorithm detects a data race in the computation of this program running serially on a given input if and only if a data race exists in the computation.

Proof: ($\Rightarrow$) Suppose ALL-SETS-SHARED declares a race between the current thread e_2 and a previous thread e_1; we show that a race between these threads indeed exists. Let d be the value of the iteration variable i in ACCESS (line 6 at the time of the race declaration. Thus, we know that e_2 is at a lock-sharing depth of at least d. Since every locker added to $lockers^{(i)}$ in line 17 consists of a thread and the depth-i lock set held by that thread, from line 8 we know that the lock-sharing depth of e_1 is also at least d and that the intersection of the depth-d lock sets of e_1 and e_2 is empty.

We now consider the depth at which e_1 and e_2 are cousins. If they are depth-d cousins, then they form a data race, since their the intersection of their depth-d lock sets is empty. If they are depth-j cousins for some $j > d$, then they also form a data race, since, with $j > d$, their depth-j lock sets are subsets of their depth-d lock sets and so do not have any locks in common. It now suffices to show that the accesses cannot be depth-j cousins for any $j < d$. Suppose for contradiction that e_1 and e_2 are depth-j cousins for some $j < d$, and let p_1 and p_2 be the depth-d locked P-node among the ancestors of e_1 and e_2, respectively. Then $p_1 \neq p_2$, since otherwise e_1 and e_2 would be depth-k cousins for some $k \geq d$. As no nodes in an SP-tree at the same lock-sharing depth can be descendents of one another, p_1 and p_2 root non-overlapping subtrees. Consider what happens to $lockers^{(d)}[l]$ in lines 2–3 at the first access to l after the subtree rooted by p_1 finishes, which must occur either before or in thread e_2, since e_2 accesses l, happens after e_1, and is not a descendent of p_1. Either the access is at lock-sharing depth less than d, or its depth-d locked P-node ancestor, stored in $pstack^{(d)}$, is a P-node other than p_1, stored in $pid[lockers^{(d)}[l]]$: in both cases, $lockers^{(d)}[l]$ is cleared in line 3. Thus, by the time e_2 runs, the locker with e_1 is not in $lockers^{(d)}[l]$; this contradicts our premise that a race between e_1 and e_2 had been declared, implying that the cousinhood between e_1 and e_2 is at least as deep at d and so form a data race, as shown above.

($\Leftarrow$) Assuming a data race exists in a computation, we show that a data race is reported. Choose two threads e_1 and e_2 such that e_1 is the last thread before e_2 in

[3]We give direct proofs of the correctness and performance bounds of ALL-SETS-SHARED—they are not overly complicated—even though it is possible to provide more succinct and modular proofs based on the correctness and bounds of ALL-SETS (Theorems 3.1 and 3.2). For an idea of how this might be done, see the proofs for BRELLY-SHARED (Corollary 4.7 and Theorem 4.8) in Section 4.3, which are based on the proofs for BRELLY.

the serial execution which has a data race with e_2. Suppose these two threads are depth-d cousins, which means they are both at lock-sharing depth at least d, and let $H_1^{(d)}$ and $H_2^{(d)}$ be their depth-d locks sets, respectively. Since e_1 and e_2 form a data race, we know $H_1^{(d)} \cap H_2^{(d)} = \{\}$.

We first show that immediately after e_1 executes, $lockers^{(d)}[l]$ contains some thread e_3 that races with e_2. If $\langle e_1, H_1^{(d)} \rangle$ is added to $lockers^{(d)}[l]$ in line 17, then e_1 is such an e_3. Otherwise, the *redundant* flag must have been set in line 15, so there must exist a locker $\langle e_3, H_3^{(d)} \rangle \in lockers^{(d)}[l]$ with $e_3 \parallel e_1$ and $H_3^{(d)} \subseteq H_1^{(d)}$. Thus, by pseudotransitivity (Lemma 2.2), we have $e_3 \parallel e_2$. Moreover, since $H_3^{(d)} \subseteq H_1^{(d)}$ and $H_1^{(d)} \cap H_2^{(d)} = \{\}$, we have $H_3^{(d)} \cap H_2^{(d)} = \{\}$, and therefore e_3, which belongs to $lockers^{(d)}[l]$, races with e_2.

To complete the proof, we now show that the locker $\langle e_3, H_3^{(d)} \rangle$ is not removed from $lockers^{(d)}[l]$ between the times that e_3 and e_2 are executed. A locker can be removed in either line 13 or line 3. Suppose for contradiction that $\langle e_4, H_4^{(d)} \rangle$ is a locker that causes $\langle e_3, H_3^{(d)} \rangle$ to be removed from $lockers^{(d)}[l]$ in line 13. Then, we must have $e_3 \prec e_4$ and $H_3^{(d)} \supseteq H_4^{(d)}$, and by Lemma 2.1, we have $e_4 \parallel e_2$. Moreover, since $H_3^{(d)} \supseteq H_4^{(d)}$ and $H_3^{(d)} \cap H_2^{(d)} = \{\}$, we have $H_4^{(d)} \cap H_2^{(d)} = \{\}$, contradicting the choice of e_1 as the last thread before e_2 to race with e_2.

Suppose for contradiction that some access between e_3 and e_2 causes $lockers^{(d)}[l]$ to be cleared in line 3. This means the computation has left the subtree rooted by e_3's depth-d locked P-node ancestor, and therefore also the subtree rooted by e_1's depth-d locked P-node ancestor, as e_1 executed before or is equal to e_3. Hence, e_1 and e_2 cannot be depth-d cousins, a contradicting the definition of d.

Therefore, thread e_3, which races with e_2, still belongs to $lockers[l]$ when e_2 executes, and so lines 7–9 report a race. ∎

The following theorem shows that All-Sets-Shared runs a factor of k slower, and uses a factor of k more space, than All-Sets, where k is the maximum number of simultaneously held locks.

Theorem 3.7 Consider a Cilk program with locks and critical sections containing parallelism, restricted as described above, that, on a given input, executes serially in time T, references V shared memory locations, uses a total of n locks, and holds at most k locks simultaneously. The All-Sets-Shared algorithm checks this computation for data races in $O(kLT(k+\alpha(V,V)))$ time and $O(k^2LV)$ space, where L is the maximum of the number of distinct lock sets used to access any particular location.

Proof: We first prove bound on space. Space usage is dominated by the lists of lockers recorded for each location. Observe that no two lockers in $lockers^{(i)}[l]$ for a given depth i and location l have the same lock set, because the logic in lines 11–15 ensures that if $H_i^{(=)} H'$, then locker $\langle e, H_i^{()}$ either replaces $\langle e', H' \rangle$ (line 13) or is considered redundant (line 15). Thus, there are at most L lockers in the list $lockers^{(i)}[l]$ for a given i. The maximum lock-sharing depth of any access is at most k, so there are at most k lists of lockers, and so the total number of lockers for a single location is at most kL. Each lock set takes at most $O(k)$ space, so the lockers for a single location take at most $O(k^2L)$ space, for a total space of $O(k^2LV)$.

We now prove the bound on running time. Since each each lock set contains at most k locks and the maximum value of D is at most k, each instance of LOCK and UNLOCK run in $O(k^2)$ time. Apart from the SP-BAGS logic, each instance of SPAWN and SYNC runs in $O(1)$ time. In ACCESS, lines 1–4 runs in $O(k)$ time, and, in each of the at most k iterations of lines 6–17, the algorithm performs at most $2L$ series/parallel tests (line 8 and lines 12 and 14) and $2L$ set operations (lines 8, 12, and 14), since the there are at most L lockers in each $lockers^{(i)}$. Each series/parallel test takes amortized $O(\alpha(V,V))$ time, and each set operation takes $O(k)$ time. Therefore, the ALL-SETS-SHARED algorithm runs in $O(kLT(k+\alpha(V,V)))$ time. ∎

Chapter 4

Data-race detection with the umbrella locking discipline

The ALL-SETS algorithm from Chapter 3 seems to be practical for programs in which L—the maximum number of combinations of locks held during accesses to any particular location—is a small constant. Such programs are common, since often a particular data structure is consistently accessed using the same lock or set of locks. When L is is not a small constant, however, ALL-SETS may be impractical. In this chapter, we see how the problem of data-race detection can be simplified by the adoption of the "umbrella locking discipline," and we present the BRELLY and BRELLY-SHARED algorithms for detecting violations of the discipline. BRELLY detects violations of the discipline in $O(kT\,\alpha(V,V))$ time using $O(kV)$ space on a Cilk program that runs serially on a given input in time T, uses V shared memory locations, and holds at most k locks simultaneously. BRELLY-SHARED, an extension of BRELLY which handles programs with critical sections containing parallelism, runs only a factor of k slower and uses only a factor of k more space.

These improvements in performance come at the cost of flexibility and precision, since the umbrella discipline precludes some race-free locking protocols as well as data races. Specifically, it requires that within every parallel subcomputation, each location is protected by a unique lock. While the umbrella discipline is more flexible than similar locking disciplines proposed in the context of race detection, it is more restrictive than ALL-SETS, which may be considered to enforce an ideal locking discipline that only disallows data races.

This chapter is organized as follows. Section 4.1 defines and discusses the umbrella locking discipline. Section 4.2 presents the BRELLY algorithm for detecting violations of the umbrella discipline, and Section 4.3 presents BRELLY-SHARED, an extension of BRELLY which correctly handles critical sections containing parallelism. Finally, Section 4.4 presents several heuristics that conservatively try to determine when a violation of the umbrella discipline is in fact caused by a data race. Such heuristics seem to be important in the practical use of BRELLY and BRELLY-SHARED for data-race detection.

Sections 4.1 and 4.2 are based largely on joint work published in [3].

4.1 The umbrella locking discipline

In this section, we introduce and discuss the "umbrella locking discipline," violations of which are detected by the Brelly and Brelly-Shared algorithms (Sections 4.2 and 4.3, respectively). The umbrella discipline requires that within each parallel subcomputation, all accesses to any particular location are protected by a single lock; parallel subcomputations in series with each other may use different locks to protect the same location. By detecting violations of this discipline rather than data races directly, we gain efficiency at the cost of precision and flexibility, since the discipline precludes some race-free locking protocols as well as data races.

Most programs do not use many different combinations of locks to access the same locations. Typically, there is a lock associated with each element of a shared data structure which is used to protect parallel accesses to that element against each other. Or there are a fixed number of global locks, which are used in a regular way to make parallel subcomputations atomic with each other. In these cases, the L in the performance bounds shown for All-Sets in Theorem 3.2 will be a small constant and the algorithm will run with roughly a constant factor slowdown, and roughly a constant factor blowup in space, as compared to the running time and space usage of the serial computation. There are some programs, however, which use many lock sets while accessing the same locations—i.e. L is not a small constant—and for which All-Sets may be unacceptably inefficient. For example, consider a graph algorithm that performs operations on arbitrary graph edges in parallel: at each edge operation, the algorithm acquires the locks associated with the nodes at both the head and tail of the edge and updates the two nodes. In the worst case, a node of degree d will be accessed with d different lock sets; in a dense graph, L will be linear with the size of the graph. (See Chapter 5 for empirical results indicating that the efficiency of All-Sets indeed varies directly with L, with performance slowing down significantly as L grows with input size.)

How can we make race detection more efficient when L is not a small constant? The Brelly algorithm presented in the next section runs in $O(kT\,\alpha(V,V))$ time and uses $O(kV)$ space, for a Cilk computation that runs serially in T time, uses V space, and holds at most k locks simultaneously. These bounds are a factor of L better than those for All-Sets, and depend only on k, which is almost never more than 2 or 3. The improvement in performance come at a cost, however. Rather than detecting data races directly, Brelly only detects violations of a locking discipline that precludes data races, and also some other race-date locking protocols. We now define this discipline, called the "umbrella locking discipline."

The umbrella locking discipline requires all accesses to any particular location within a given parallel subcomputation to be protected by a single lock. Subcomputations in series may each use a different lock, or even none, if no parallel accesses to the location occur within the subcomputation. This discipline can be defined precisely in terms of the parse tree of a 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 4-1. Umbrellas are always considered with respect to accesses to a single location l.

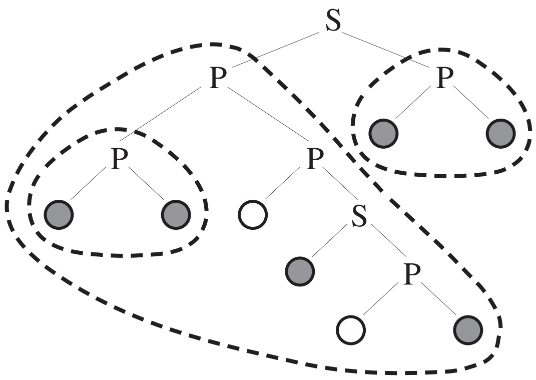


Figure 4-1: Three umbrellas of accesses to a location l. In this parse tree, each shaded leaf represents a thread that accesses l. Each umbrella of accesses to l is enclosed by a dashed line.

The ***umbrella locking discipline*** requires that, in every umbrella of accesses to a location l be "protected," that is, there be some lock that protects all the accesses in the umbrella against each other. In other words, within each umbrella of accesses to a location l, all threads must agree on at least one lock to protect their accesses to l. Under the assumption that critical sections do not contain parallelism, the notion of protected umbrellas can be formally defined as followed: an umbrella of accesses to l is ***protected*** if the lock set of these accesses is nonempty, and ***unprotected*** otherwise. (Recall from Chapter 2 that the lock set of several accesses is the intersection of their respective individual lock sets.) Since instances of a lock are never shared by parallel accesses, any lock held by all the accesses in an umbrella will protect them against each other.

If critical sections can contain parallelism, the definition of a protected umbrella needs to be refined to allow for the possibility that a lock held by all accesses in an umbrella may not protect the umbrella, since the same instance of the lock may be shared by some or all of the accesses. Specifically, we say that an umbrella of accesses to l is ***protected*** if the lock set of these accesses contains some lock that is *not* held across any pair of parallel accesses to l in the umbrella, and ***unprotected*** otherwise. Equivalently, we can say that an umbrella is protected if the lock set of its accesses contains some lock that is *not* held across the entire umbrella, and it does not contain an unprotected umbrella; it is unprotected otherwise.

The next theorem implies that adherence to the umbrella discipline precludes data races from occurring.

Theorem 4.1 A Cilk computation with a data race violates the umbrella discipline.

Proof: Any two accesses involved in a data race must have a P-node p as their least common ancestor in the parse tree, because they operate in parallel. If the lock sets of the two accesses are disjoint, then p roots an unprotected umbrella. Otherwise, any locks held in common by the accesses must be shared by them, i.e. held across p, and so p roots an unprotected umbrella. ■

8iuThe umbrella discipline can also be violated by unusual, but data-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 this and other unusual locking protocols (which may or may not be as silly), BRELLY checks programs asymptotically much faster than ALL-SETS.

How might the umbrella discipline be used in practice? First, a programmer may choose to adopt the umbrella discipline as a good programming practice, and write code accordingly. Then data races can be detected using the BRELLY (or BRELLY-SHARED) algorithm. Even umbrella violations not resulting from races would be considered breaches of programming practice and should be fixed. Alternatively, a programmer can use the BRELLY algorithm to detect violations of the discipline without trying to follow it *a priori*, and then manually determine whether violations are caused by data races, ignoring those that are not.

4.2 The BRELLY algorithm

We now present the BRELLY algorithm for detection violations of the umbrella discipline in Cilk computations with locks and serial critical sections. On a program running serially on a given input in time T using V shared memory locations, BRELLY detects umbrella discipline violations in $O(kT\,\alpha(V,V))$ time using $O(kV)$ space, where k is the maximum number of locks held simultaneously. After describing the algorithm and giving an example of its execution, we show that it correctly detects umbrella discipline violations and prove these performance bounds.

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 4-2 gives pseudocode for BRELLY. Like the SP-BAGS algorithm, 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 in $O(\alpha(V,V))$ time. 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: $accessor[l]$, which stores for each location l a thread that performed an access to that location; and $locks[l]$, which stores the lock set of that access. (Exactly which thread is stored in $accessor[l]$ is explained below.) 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.

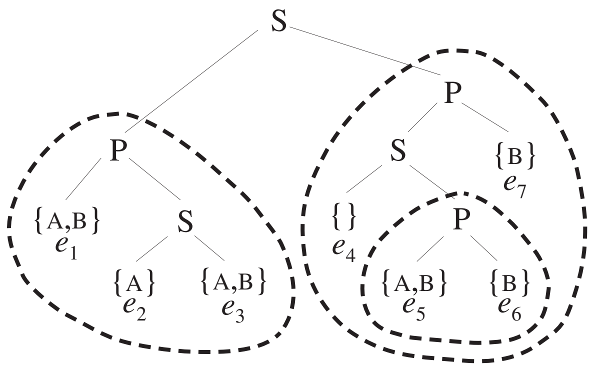
Unlike the ALL-SETS algorithm, BRELLY keeps only a single lock set, rather than a list of lock sets, for each shared memory location. For a location l, each lock in $locks[l]$ potentially belongs to the lock set of the largest umbrella of accesses to l that includes the current thread. The BRELLY algorithm tags each lock $h \in locks[l]$ with two pieces of information: a thread $nonlocker[h]$ and a flag $alive[h]$. Each of these tags

```
ACCESS(l) in thread e with lock set H
if accessor[l] ≺ e
   then ▷ serial access
        locks[l] ← H, leaving nonlocker[h] with its old
           nonlocker if it was already in locks[l] but
           setting nonlocker[h] ← accessor[l] otherwise
        for each lock h ∈ locks[l]
             do alive[h] ← TRUE
        accessor[l] ← e
   else ▷ parallel access
        for each lock h ∈ locks[l] − H
             do if alive[h] = TRUE
                   then alive[h] ← FALSE
                        nonlocker[h] ← e
        for each lock h ∈ locks[l] ∩ H
             do if alive[h] = TRUE and nonlocker[h] ∥ e
                   then alive[h] ← 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 ∈ H ∩ locks[l]
                     do report access to l without h
                          by nonlocker[h]
```

Figure 4-2: The BRELLY algorithm. While executing a Cilk program in serial depth-first order, at each access to a shared memory location l, the code shown is executed. Not shown are the updates to H, the set of currently held set of locks, which occur whenever locks are acquired or released. To determine whether the currently executing thread is in series or parallel with previously executed threads, BRELLY uses the SP-bags data structure from [9].

is associated with the *entry* of a lock in *locks*$[l]$ for some location l; no tag is associated with a lock globally, across all locations. The thread *nonlocker*$[h]$ is a ***nonlocker*** of l with respect to lock h, that is, a thread which 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 specify which threads are involved, the algorithm ***kills*** a lock h by setting *alive*$[h] \leftarrow$ 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 the benefit of future accesses. The way any particular access to l is handled depends on whether it is logically in series or in parallel with the thread in *accessor*$[l]$ at the time. If *accessor*$[l] \prec e$, we say the access is a ***serial access***, and the algorithm



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

Figure 4-3: An example 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. $\{\mathrm{A}, \mathrm{B}\}$). 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.

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 discovers in line 14 that there are no locks left alive in *locks*$[l]$ after a parallel access, it has discovered an unprotected umbrella, and it reports a discipline violation in lines 15–17.

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 which accessed l without each of these locks.

Figure 4-3 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, lock A is killed and *nonlocker*$[\mathrm{A}]$ 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*[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.

The following two lemmas are helpful in proving the correctness of BRELLY.

Lemma 4.2 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. 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 if $e' \parallel e$, by pseudotransitivity we would have $e'' \parallel e$, a contradiction. ∎

Lemma 4.3 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 line 10, which sets *nonlocker*$[h] \leftarrow e$. This update only occurs when the current thread e does not hold lock h (line 7). The second case is when *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*$[l]$ will be the initial thread. Otherwise, *locks*$[l]$ will be the set of locks held during an access to l in *accessor*$[l]$, since *locks*$[l]$ and *accessor*$[l]$ are always updated together to the current lock set H and current thread e, respectively, during a serial access (lines 2–5). 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. ∎

We now show that BRELLY correctly detects violations of the umbrella discipline.

Theorem 4.4 Consider a Cilk program with locks and serial critical sections. The BRELLY algorithm detects a violation of the umbrella discipline in a computation of this program running serially 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 $p = \text{LCA}(e_0, e)$ 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 4.3, w accesses l without the lock h. Assume for the purpose of contradiction that w is not a descendant of p. Then, we have $\text{LCA}(w, e_0) = \text{LCA}(w, e')$, 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 4.2, 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 4.2, 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 $p' = \text{LCA}(accessor[l], p)$. 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 which 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 2.2), $e' \parallel e_{left}$ 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. ∎

We now show the performance bounds for BRELLY.

Theorem 4.5 On a Cilk program with locks and critical sections without parallelism, which on a given input 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 $|H| \le k$. Hence, the total space is $O(kV)$.

Each loop in Figure 4-2 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. ∎

4.3 The Brelly-Shared algorithm

In this section we present Brelly-Shared, an extension of Brelly which correctly handles programs with critical sections that contain parallelism—as specified in Section 3.2. After describing how the algorithm works, we prove its correctness and performance bounds, which are a factor of k larger than those for Brelly in both time and space. Rather then directly proving the Brelly-Shared detects discipline violations, we show that it essentially simulates the original Brelly algorithm at each lock-sharing depth at each access, and that it does so correctly. This proof is an example of a general approach to extending data-race detection algorithms based on the SP-tree model to handle critical sections containing parallelism.

The basic approach of Brelly-Shared mirrors that of All-Sets-Shared: just as All-Sets-Shared simultaneously runs an instance of the original All-Sets for each depth of cousinhood at each access, comparing only locks sets and lockers of the same lock-sharing depth, so Brelly-Shared simultaneously runs an instance of the original Brelly for each depth at each access. In Brelly, for each location l, there is a single $accessor[l]$ variable, which holds the last serial access to l; and a single $locks[l]$ variable, which holds the locks that may protect the largest umbrella including the current thread. In Brelly-Shared, there are several instance of both of these variables, indexed by lock-sharing depth i: $accessor^{(i)}[l]$ holds the last serial access *among the depth-i cousins of the current thread*, and $locks^{(i)}[l]$ keeps the locks which may protect the largest umbrella *among these depth-i cousins*. Consider the SP-tree in Figure 3-5 for intuition: imagine the original Brelly running on the entire tree using depth-0 lock sets, then on the subtree rooted by the root's right child using depth-1 lock sets, and finally on the subtree rooted by $\text{LCA}(e_3, e_4)$ using depth-2 lock sets.

Besides the multiple *accessor* and *locks* variables for each location, Brelly-Shared maintains the following global variables as in All-Sets-Shared: D, the current lock-sharing depth; $H^{(i)}$ for $0 \leq i \leq D$, the current depth-based lock sets; and $pstack^{(i)}$ for $1 \leq i \leq D$, the IDs of the locked P-nodes among the current thread's ancestors. For each location l, the algorithm keeps the lock-sharing depth of the last access to l in $locks\text{-}depth[l]$, and IDs of the locked P-nodes among that access's ancestors, oldest to most recent, in $pid^{(i)}[l]$ for $1 \leq i \leq locks\text{-}depth[l]$.

The logic for accesses in Brelly-Shared is shown in Figure 4-4; the logic for acquiring and releasing locks, and for parallel control commands, is the same as in All-Sets-Shared (Figure 3-6). In Access, lines 1–5 updates the *accessor* and *locks* variables according to the locked P-node ancestors of the current access, clearing $locks^{(i)}[l]$ and setting $accessor^{(i)}[l]$ to the ID of the initial thread (which logically precedes all other threads) if the subtree rooted by the locked P-node recorded in $pid^{(i)}[l]$ has already completely executed. Lines 7–24 then execute the original Brelly logic at each depth, comparing against $accessor^{(i)}[l]$ and intersecting the current depth-i lock set into $locks^{(i)}[l]$ at each iteration.

Instead of directly proving that Brelly-Shared detects umbrella discipline violations, we show that Brelly-Shared correctly performs the original Brelly logic for each subtree rooted by a locked P-node, using the lock sets appropriate to that

```
ACCESS(l) in thread e
for i ← 1 to D
    do if pstack^(i) ≠ pid^(i)[l] or i > locks-depth[l]
        then locks^(i)[l] ← {}
             accessor^(i)[l] ← ID of initial thread
             pid^(i)[l] ← pstack^(i)
locks-depth[l] ← D
for i ← 0 to D
    do if accessor^(i)[l] ≺ e
        then ▷ serial access
             locks^(i)[l] ← H, leaving nonlocker[h] with its old
               nonlocker if it was already in locks^(i)[l] but
               setting nonlocker[h] ← accessor^(i)[l] otherwise
             for each lock h ∈ locks^(i)[l]
                 do alive[h] ← TRUE
             accessor^(i)[l] ← e
        else ▷ parallel access
             for each lock h ∈ locks^(i)[l] − H
                 do if alive[h] = TRUE
                       then alive[h] ← FALSE
                            nonlocker[h] ← e
             for each lock h ∈ locks^(i)[l] ∩ H
                 do if alive[h] = TRUE and nonlocker[h] ∥ e
                       then alive[h] ← FALSE
             if no locks in locks^(i)[l] are alive (or locks^(i)[l] = {})
               then report violation on l involving
                       e and accessor^(i)[l]
                    for each lock h ∈ H ∩ locks^(i)[l]
                        do report access to l without h
                             by nonlocker[h]
```

Figure 4-4: The BRELLY-SHARED algorithm. While executing a Cilk program in serial depth-first order, at each access to a shared memory location l, the code shown is executed. The logic for LOCK and UNLOCK, and for SPAWN, SYNC, and RETURN, is the same as that in ALL-SETS-SHARED (Figure 3-6).

subtree. Recall that the global data structures of the algorithm are correctly maintained as described (Lemma 3.5).

Theorem 4.6 Let p be a depth-d locked P-node in the SP-tree of a Cilk computation with any parallelism within crtical sections restricted as described in Section 3.2. The BRELLY-SHARED algorithm, when run on this computation, executes the logic of BRELLY on the subtree rooted by p as if the subtree represented a self-contained computation, using the depth-d lock sets of the accesses in the subtree.

Proof: Pick an arbitrary location l and consider BRELLY-SHARED's actions concerning l during the execution of p's subtree. If there are no accesses to l in p's subtree, then the algorithm does nothing, just as BRELLY would do nothing in a tree with no accesses to l. Now suppose there are accesses to l in the subtree, and let e be the first such access in the serial depth-first execution. Consider what happens in the lines 1–5 of ACCESS(l) during e, in the iteration when $i = d$. Since this is the first access to l in p's subtree, the P-node recorded in $pid^{(d)}$, if $locks\text{-}depth[l]$ is not less than D, will not be p, and so $locks^{(d)}[l]$ will be set to $\{\}$, $accessor^{(i)}[l]$ to the ID of the initial thread, and $pid^{(d)}[l]$ to p. Note that $locks^{(d)}[l]$ and $accessor^{(d)}[l]$ have been initialized as they would have been in the beginning of BRELLY, were it run on the subtree rooted by p. Now, $pid^{(d)}[l]$ will not be set to another value, and so $locks^{(d)}[l]$ and $accessor^{(d)}[l]$ will not be reset, until the execution finishes all the thread in p's subtree; in the meantime, BRELLY-SHARED will update $locks^{(d)}[l]$ and $accessor^{(d)}[l]$ exactly according to the logic of BRELLY (lines 8–24 are exactly analogous to the code for ACCESS(l) in BRELLY), except using depth-d lock sets at each access. Thus, the theorem holds. ∎

Corollary 4.7 The BRELLY-SHARED algorithm detects a violation of the umbrella discipline in a computation of a Cilk program with locks running serially on a given input if and only if a violation exists. ∎

The following theorem shows that BRELLY-SHARED runs a factor of k slower, and uses a factor of k more space than BRELLY, where k is the maximum number of locks held simultaneously.

Theorem 4.8 On a Cilk program which on a given input executes serially in time T, uses V shared memory locations, and holds at most k locks simultaneously, the BRELLY-SHARED algorithm runs in $O(k^2 T\,\alpha(V,V))$ time and $O(k^2 V)$ space.

Proof: Since the lock-sharing depth of any access is at most k, the algorithm keeps at most k times the number of *locks* and *accessor* entries per location, and iterates at most k times through lines 7–24 at each access, performing at most k times as many series/parallel checks as BRELLY. Thus, the bounds follow from the bounds shown for BRELLY (Theorem 4.5). ∎

4.4 Data-race detection heuristics for Brelly and Brelly-Shared

A user may adopt the umbrella discipline as a good programming practice and then use Brelly or Brelly-Shared to find violations of the discipline, which are always considered undesirable. However, some users may be concerned primarily with data races outright. In this section we outline several heuristics to improve the usefulness of Brelly's output, by causing it to report straightforward data races and hide non-race violations—i.e. unprotected umbrellas which do not contain data races—whenever possible. These heuristics are conservative: they never hide violations that are caused by data races unless a related data race has already been reported. For simplicity, we discuss the heuristics in the context of Brelly; they extend in the natural way to Brelly-Shared.

Before considering the heuristics themselves, recall how Brelly reports violations. As given in Figure 4-2, it reports violations by specifying the memory location, the current access, the *accessor* access, and the *nonlocker* access of each lock that was held during both the the current access and the *accessor*. In identifying an access, the original string of code that was the source of that access can be printed: the filename, line number, and the variable name used in the code. The *nonlocker* accesses show the user why the locks held by both the current access and the one in *accessor* do not protect the umbrella rooted at their least common ancestor. Unfortunately, it is not always easy or possible to determine from such a report where the data race is, or whether there is a data race at all, which is why we would like to conservatively report a violations as being between exactly two accesses—a straightforward data race—whenever possible.

Suppose Brelly detects a methodology violation during an access to location l in thread e which holds the lock set H: $accessor[l] \parallel e$ and there are no alive locks in $locks[l]$. The following heuristics can be used:

1. Report a data race between e and $accessor[l]$ if $H \cap locks[l] = \{\}$. This is already the logic of lines 15–17 in Figure 4-2, since in such a case only two threads will be involved in the violation, clearly indicating a data race.

2. If there is exactly one lock c in H and $c \in locks[l]$, report a data race between c and $nonlocker[c]$ if $nonlocker[c] \parallel e$; otherwise ignore the violation as an umbrella violation without any data race. This heuristic is only correct if Brelly is slightly modified: line 10 should set $nonlocker[c] \leftarrow e$ only if e is in series with the existing $nonlocker[c]$. This logic ensures that the *nonlocker* of each lock in $locks[l]$ is the "most-parallel" access to l without that lock, that is, the nonlocker which any future thread will be logically in parallel with if it is in parallel with any such nonlocker.

3. If a data race has been reported since the most recent serial access (i.e. since $accessor[l]$ was last updated), hide the violation unless there is a data race between e and $accessor[l]$ (determined with heuristics 1 and 2). Although this

heuristic might hide data races, this seems acceptable since a data race in the same umbrella has already been discovered and reported. This heuristic requires a per-location "data-race detected" flag to be set whenever a data race is detected and reset at serial accesses.

4. If a data race has already been reported between the source code strings that caused the accesses in e and $accessor[l]$ (regardless of whether it was the same memory location), hide the violation. In general, hide the violation if a data race between the source code strings of any pair of accesses involved, including the relevant nonlockers, has already been reported. This requires keeping track of the races reported, which can be done efficiently with a hash table.

5. If heuristics 1, 2, and 4 do not apply, hide the violation if a violation involving the same set of accesses (including the nonlockers) has already been reported. This can be done with a hash table, as in heuristic 4. A more aggressive and efficient version of this heuristic, which may cause data races to be nonconservatively hidden by previous false violations, is to hide the violation if any violation with the same "endpoints" —i.e. the current thread and *accessor*—has been reported.

6. Summarize violations at the end, leaving out violations that can be been hidden based on data races found subsequently (a la heuristic 4). Also, while summarizing, violations can be sorted by source code string as an aid to the user when tracking down bugs.

Some highly preliminary testing has been done with these heuristics. On a particular computation (Chapter 5's `maxflow` running on a typical input), BRELLY, with no heuristics, reported 65 violations. With heuristics 1-4 and the aggressive, nonconservative version of heuristic 5 enabled, BRELLY reported 25 straightforward data races and 6 unprotected umbrellas. Of the unprotected umbrellas, the 2 that contain data races would have been correctly hidden by heuristic 6 (since the data races were subsequently reported separately) and the other 4 are easily checked manually by the user.

To give a sense of how many of these apparent data races and violations are infeasible, we note that there is only one a single bug related to data races in `maxflow`: one line of code, which wrote to shared memory, was mistakenly put just after the end of a crtical section instead of within it. This bug leads to a handful of feasible data races, since the unprotected write in the misplaced line races with several other accesses. The large majority of the data races and umbrella discipline violations reported in this example are, however, infeasible. They result from the practice of memory publishing (a linked-list work queue is used extensively in `maxflow`), which is discussed in the conclusion (Chapter 7).

Chapter 5

Empirical comparison of All-Sets and Brelly

In this section, we compare the empirical performance of All-Sets and Brelly on a number of programs and inputs, based on preliminary tests with four Cilk programs that use locks and several inputs. These initial findings strongly indicate that our performance bounds for the algorithms are at least roughly reliable. Specifically, we see that L, the factor by which Brelly's performance bounds are better than All-Sets's, is predictive: if a program always accesses a particular location with the same lock (so L is a small constant), All-Sets performs as fast as Brelly; when the number of lock sets per location grows with input size (so L grows with input size), Brelly significantly outperforms All-Sets. In addition, we see that the factor by which Brelly slows down the original program's computation is a constant independent of input size, whereas the factor by which All-Sets slows down a program can grow with input size. Finally, we give an initial indication of how many nonrace umbrella violations Brelly typically reports.

Our tests were done using the Nondeterminator-2, a new version of the Nondeterminator (see Chapter 2) currently under development. The implementations of All-Sets and Brelly are not optimized, and so better performance than what we report here is likely to be possible.

According to Theorem 3.2, the factor by which All-Sets slows down a program compared to its original running time is roughly $O(kL)$, 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 4.5, the slowdown factor for Brelly is about $O(k)$. 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, not counting the implicit fake R-LOCK used by the detection algorithms:

`maxflow`: A maximum-flow code based on Goldberg's push-relabel method [11]. Each vertex in the graph contains a lock. Parallel threads perform simple

This chapter is based on joint work published in [3].

	parameters			time (sec.)			slowdown	
program	input	k	L	original	ALL-SETS	BRELLY	ALL-SETS	BRELLY
`maxflow`	sparse 1K	2	32	0.05	30	3	590	66
	sparse 4K	2	64	0.2	484	14	2421	68
	dense 256	2	256	0.2	263	15	1315	78
	dense 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`	iteration 1	2	65	1.2	109	45	91	37
	iteration 2	2	94	1.0	179	45	179	45
	iteration 5	2	168	2.8	773	94	276	33
	iteration 13	2	528	9.1	13123	559	1442	61

Figure 5-1: Timings of our implementations on a variety of programs and inputs. 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.

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" 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 [5, 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`: A 3-dimensional radiosity renderer running on a "maze" scene. The original 75-source-file C code was developed in Belgium by Bekaert et. al. [2]. We used Cilk to parallelize its scene geometry calculations. Each surface in the scene has its own lock, as does each "patch" of the surface. In order to lock a patch, the surface lock must also be acquired, 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 5-1 shows the 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 due to larger inputs, while the slowdown factor of BRELLY is constant, independent of L and input size. 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.[1]

Although we have not done formal testing, it appears that the number of nonrace umbrella violations reported by BRELLY can be significant, even overwhelming, in some cases. These violations are typically infeasible, being caused by the common practice of memory publishing (see discussion in Chapter 7). For an example one such case, see the discussion of `maxflow` at the end of Section 4.4. Programs that do not publish memory seem to naturally obey the umbrella discipline much more readily, with no major cause of discipline violations.

[1]For a production debugging tool, the implementations of these and the other algorithms in this thesis would need to be highly optimized, since they instrument every access to shared memory. We expect that one key optimization is to handle reads and writes separately with code specific to each case and without the convenience of a fake read lock. Is it be possible to implement the algorithms efficiently enough to be general purpose, comparable to SP-BAGS in efficiency when no locks are used?

Chapter 6

Data-race detection in computations with guard statements

In this chapter, we discuss data-race detection in programs that use a proposed "guard statement" language construct, instead of locks, for atomicity. The guard statement allows users to specify atomicity at a higher level than with locks, and with built-in structure. Happily, because of the more restrictive semantics of guard statements, detecting data races in programs with guard statements is easier than detecting data races in those with locks. We present the REVIEW-GUARDS algorithm, which, on a Cilk program that runs serially on a given input in time T, uses V shared memory locations, and guards at most k memory blocks simultaneously, detects data races in $O(T(\lg k + \alpha(V, V)))$ time using $O(V + k)$ space. The REVIEW-GUARDS-SHARED algorithm, an extension of REVIEW-GUARDS which correctly handles critical sections containing parallelism, achieves the same asymptotic performance bounds.

Our discussion of the guard statement is not intended to be definitive: the proper way to provide structure atomicity is left as an open question. The guard statement proposed in this chapter does, however, touch on several key design choices and provides a basis for the REVIEW-GUARDS algorithms, which show that data races can be detected efficiently in programs with a reasonable form of higher-level structured atomicity.

This chapter is organized as follows. In Section 6.1, we propose a syntax and semantics for the guard statement, with consideration of how the new language construct might be implemented at runtime. In Section 6.2, we compare the guard statement with locks, showing that, although the data-race detection algorithms for locks can be modified to work with guard statements, new algorithms specific to guard statements can give better performance. Accordingly, we present the REVIEW-GUARDS and REVIEW-GUARDS-SHARED algorithms in Sections 6.3 and 6.4, respectively.

The proposed guard statement and implementation ideas in Section 6.1 were developed jointly with Charles E. Leiserson, Mingdong Feng, and other members of the Cilk group at MIT.

6.1 The guard statement for providing structured atomicity

In this section, we propose a "guard statement" language construct for providing structured atomicity. We first explain some of the disadvantages of locks, motivating the need for a way to specify atomicity at a higher level. We then specify and syntax and semantics of the guard statement, showing how it might be implemented at runtime. In order to achieve an efficient implementation, we then redefine guard statements to include "same-start semantics": parallel access are mutually atomic only if they are guarded using memory blocks with the same start address. We argue that these semantics can be implemented efficiently. Finally, we discuss the limitations on expressibility imposed by guard statements with these restricted semantics.

Using locks to provide atomicity has two major disadvantages. First, locks require users to allocate lock variables explicitly in their code, which can be a nuisance and possibly a serious hit on memory usage, as, for example, in the case of a large array with a lock associated with each element. Second, and more importantly, the flexibility of locks, which can be acquired and released according to whatever protocol (or lack of protocol) a user chooses to follow, can easily lead to convoluted or incorrect code. For instance, deadlock, in which a program comes to a standstill because each thread waits for another thread to release a lock in a circular dependency, is a common problem.

These problems with locks are not insurmountable: deadlock, for one, can be avoided by the standard discipline of always acquiring locks in a fixed global order. Nonetheless, some languages limit the flexibility of locks—for instance, by allowing at most one lock to be held at a time. Other languages hide locks from the user altogether and provide atomicity with higher-level language constructs. Java, for instance, does not provide explicit locks but allows methods of an object class to be denoted as "synchronized"; such methods, if invoked in parallel for a single object, operate atomically with respect to each other. Java also provides a separate language construct, with the syntax

$$\texttt{synchronized(}\ \textit{object}\ \texttt{)}\ \{\ \textit{statements}\ \}$$

that implicitly acquires the lock associated with *object* (all Java objects have hidden locks associated with them), executes the code in *statements*, and then releases the lock. These Java language features for atomicity allow users to ignore the details of locks and reason about atomicity at a higher level.

The current version of Cilk, version 5.1, supports atomicity exclusively through locks. An alternative mechanism for atomicity, which we propose here, is a ***guard statement*** language construct which allows the user to indicate that a block a code should be executed atomically *on a specific range of memory.* Consider the following syntax:

$$\texttt{guard(}\ \textit{block}_1; \textit{block}_2; \ldots\ \texttt{)}\ \{\ \textit{statements}\ \}$$

Between the curly braces, *statements* is arbitrary Cilk code that does not contain another guard statement; this code is to be executed normally, except that accesses to the specified memory blocks will be guarded. The one or more memory blocks to be guarded are specified in a semicolon-delimited list between the parentheses just after the `guard` keyword, where each $block_i$ is either the memory associated with a variable, specified by the variable name; or an arbitrary array of memory, specified by a pointer and size, separated by a comma. For example, if `x` is a storage variable (of type `int`, `float`, `struct`, etc.) and `p1` and `p2` are pointers, the guard statement

```
guard(x; p1, 20; p2, n+1) {...}
```

has the semantics: "Within this body of code, make accesses to `x`, the 20 array elements beginning at location `p1`, and the `n+1` array elements beginning at `p2` atomic with respect to other parallel accesses to these locations which are also guarded."[1]

With guard statements, atomicity is well structured and does not requires the user to allocate extra data structures. Furthermore, as the syntax disallows nested guard statements, the runtime system can easily prevent deadlock by automatically acquiring any hidden locks it needs in a fixed global order. We therefore include in the specification of the guard statement the following guarantee, which an implementation must ensure: a program with guard statements but without locks will never deadlock.[2]

The guard statement does require more language support than locks, both in the compiler, which must handle the new syntax, and at runtime. To implement atomicity, the runtime system needs to allocate, acquire, and release locks behind the scenes, based on the user's guard statements. We suggest that the runtime system allocate a fixed number of hidden lock variables, and then translate each memory location which needs to be guarded into one of these locks using a hash function. When executing a guard statement, the locks associated (through the hash function) with each guarded memory location are acquired before, and released after, the execution of the code within the guard statement.

The glaring problem with this straightforward implementation is that guarding arrays of memory, such as in the example above (`p1, 20` and `p2, n+1`), is unacceptably inefficient, since one lock per element is acquired. (Acquiring a lock is typically an expensive operation, taking over a dozen cycles on a Sun UltraSPARC 1 using the compare-and-swap instruction, for instance.) There might be an efficient way of "locking" an entire block of memory without having to lock each byte individually. An interval-tree-like data structure in which intervals can be quickly added, deleted, and searched in parallel would be helpful, and *possibly* fast enough: the runtime system could maintain the currently guarded blocks of memory as intervals in the data

[1] The sizes of variable and array elements are determined by variable and pointer types, as usual in C. Also, to be consistent with C blocks, the open and close braces surrounding *statements* are optional when guarding a single statement.

[2] The prohibition against nested guard statements may seem overly restrictive: why not allow the user to nest guard statements if he is willing to do without the guarantee of deadlock-freedom? Because with guard statements, the order in which (hidden) locks are acquired is not in the user's control, and therefore a user cannot implement application-specific deadlock-prevention protocols, as is possible with explicit locks.

structure and, when executing a guard statement, check for already guarded memory blocks that overlap with the to-be-guarded memory blocks in the structure, waiting for them to be removed before executing the body of the guard statement. In any case, we know of no such data structure.

How then should the runtime system implement guard statements? Our solution is to modify the semantics of the guard statement. Specifically, we specify ***same-start semantics***: memory accesses are atomic with each other if they are within guard statements, the locations accessed are in memory blocks specified in the guard statements, and *these blocks of memory have the same start addresses for each location*. For example, parallel threads which guard and access memory addresses 1 through 10, 1 through 5, and the single byte at address 1, are atomic with each other; threads which guard and access addresses 1 through 10, 2 through 9, and the single byte at address 5, are not. With these more limited guard-statement semantics, the runtime system need only acquire the lock associated with the first byte of each guarded memory block, meaning that only one lock per guarded block, rather per guarded location, is acquired. For convenience, we say that the ***guard address*** of a guarded memory access is the start address of the guarded memory block that contains the accessed location, and that the access is ***protected*** by the start address. The Review-Guards and Review-Guard-Shared algorithms detect data races according to these semantics.[3]

There is another potential problem with using a hash function to associate memory locations with lock variables: since the number of available locks is presumably much smaller than the size of shared memory, multiple memory locations hash to the same lock, possibly resulting in guard statements which protect unrelated blocks of memory being forced to be atomic with each other. For example, suppose one guard statement protects location l_1 and another protects a different location l_2, and that l_1 and l_2 hash to the same hidden lock. These two guard statements are atomic with each other, as they use the same lock, which, while not incorrect, may degrade performance, since the guard statements are forced to wait for each other to finish if they run simultaneously. Fortunately, the sharing of locks by different memory locations can be solved by allocating a sufficiently large number of hidden locks. The maximum number of simultaneously guarded blocks of memory during a parallel execution, and therefore the maximum number of simultaneously held lock variables, is small: it is

[3]Limiting the semantics of the guard statements is equivalent to adopting a "guarding discipline," analogous to a locking discipline, except that in this case the motivation for the discipline is to make runtime implementation more efficient, rather than to simplify race detection or avoid deadlock. We might stipulate that a program not guard, in parallel, overlapping blocks of memory with different start addresses, and then have the race detection algorithms report violations of this discipline as well as data races under the original guard semantics. Unlike with umbrella semantics, however, violations of such a guarding discipline would always be data races, since the runtime system, under the same-start semantics, in fact only locks the first byte of each guarded block.

Note that adoption of the same-start semantics leads to an awkward situation in a typical implementation: the sizes of guarded memory blocks are ignored during runtime. The sizes remain in the syntax of the guard statement, however, since they may be used by the memory-consistency algorithm in a distributed system, as suggested below in Section 6.2.

at most kP, where k is the maximum number of simultaneously guarded memory blocks in a serial execution, and P is the number of processors. Thus, if the number of available locks is N, then the probability that any given memory location will be translated by a hash function that appears uniformly random into a lock that is currently being used for a different memory location is less than kP/N. If $k \leq 4$ and $P \leq 256$—usually safe assumptions—then, with 100 KB of hidden locks, we have $kP/N \leq 1/100$, which should be small enough to make delay caused by accidentally shared locks negligible.

How severe a limitation, algorithmically, are same-start semantics for guard statements? Programs often access identical memory blocks in parallel, sometimes even performing identical operations on them: for example, global counters which are updated in parallel; nodes of a graph which are operated on atomically by various procedures; and slots of a parallel hash table. In such cases, same-start semantics are natural and sufficient. Sometimes, however, a program may access parts of a global data structure in one thread, and the entire data structure in another, parallel thread. For example, consider a table which is updated one entry at a time, or sorted all at once: the update and sort operations, if occurring in parallel, access overlapping memory locations which do not necessarily have the same start address, and so the same-start semantics are insufficient. The original semantics of guard statements, which do not require overlapping memory blocks guarded in parallel to have the same start addresses, are needed for these table operations. The straightforward implementation of those semantics mentioned above, in which one lock per guarded location is acquired, would likely be unacceptably slow, however. It seems that, apart from a fast implementation of the original guard semantics, guard statements may are not a good choice for programming these parallel table operations.

One should note that even with locks, the correct solution is not clear: if a program using locks acquires a lock associated with the entire table each time it updates an entry, then the updates would not execute simultaneously. If, on the other hand, the program acquires every entry's lock before sorting the table, it would have the same, rather serious, performance hit as the straightforward implementation of the original guard semantics. Locks are, of course, more flexible, giving the user the choice of how to provide atomicity for any given problem; guard statements, by design, limit the possibilities in the interests of simplicity.

6.2 The need for detection algorithms specific to guard statements

Do we need data-race detection algorithms specific to guard statements? Can we use the algorithms for programs with locks, perhaps with minor changes, to detect data races in programs with guard statements? In this section, we show that although the algorithms for locks can be made to work with guard statements, they are not optimal. Thus, a need exists for detection algorithms specific to guard statements, such as Review-Guards and Review-Guards-Shared, which we present in Sections 6.3

```
cilk void guard_swap(int *q1,
                     int *q2) {
  int temp;

  guard(*q1) {
    temp = *q1;
    *q1 = *q2;
    *q2 = temp;
  }
}
```

```
cilk void lock_swap(int *q1,
                    int *q2) {
  int temp;

  Cilk_lock(A);
  temp = *q1;
  *q1 = *q2;
  *q2 = temp;
  Cilk_unlock(A);
}
```

Figure 6-1: The functions `guard_swap` and `lock_swap` ostensibly do the same thing: atomically swap the integers pointed to by the parameters `q1` and `q2`. (The variable `A` is a global lock, whose declaration and initialization is not shown.) In fact, however, `guard_swap` only guards accesses to `*q1`, whereas `lock_swap` correctly protects the entire swap operation with a lock. Changing `guard(*q1)` to `guard(*q1; *q2)` in `guard_swap` would make it correctly atomic.

and 6.4, respectively.

The translation of guarded memory blocks to locks by the runtime system may seem to imply that the locking algorithms for programs with locks presented in Chapters 3 and 4 can be used directly for race detection, leaving no need for new algorithms. One might even think the guard statement to be merely a mechanism for hiding or restricting the syntax of locks, with the same underlying semantics. For example, consider the `guard_swap` and `lock_swap` functions shown in Figure 6-1, which swap a pair of integer locations, the first using a guard statement and the second with a lock. Parallel threads running `lock_swap` will be correctly atomic with each other due to the global lock `swap_lockvar`. Wouldn't parallel threads executing `guard_lock` also be atomic with each other, since the runtime system acquires the lock associated with `*q1` before swapping the values?

The answer is no. Recall that the semantics state that only the accesses to the specifically guarded memory blocks are atomic, meaning that the guarded swap operation in `guard_swap` is not atomic with itself since `*q2` is not specifically guarded. In a distributed system with software-simulated shared memory (e.g. Distributed Cilk [24, Chapter 8]), these semantics might be implemented strictly, with the memory system ensuring consistency between parallel threads during their execution, by performing expensive memory update operations between the distributed memories, only for specifically guarded memory blocks. In such a system, the value of `*q2`, which is not guarded, might not be updated between parallel threads executing `guard_swap`, leading to incorrect behavior. The memory semantics of user locks is stronger, requiring all modified locations within a critical section to be updated between parallel threads, so `lock_swap` is atomic and correct.[4] It is true that on a hardware shared-

[4]The use of locks typically requires at least release consistency [13, p. 716], in which memory

memory machine, the implementation we have suggested for guard statements would likely result in an atomic `guard_swap` as well, since a fast memory barrier can be used to update all memory after critical sections, but this would be an accident of the implementation, not an implication of the guard statement semantics. Guard statements are indeed more than syntactic sugar; they are not merely hiding the acquisition and release of locks associated with certain memory blocks.

Detecting data races in a program that uses guard statements for atomicity is not, then, immediately reducible to the problem of detecting data races in programs with locks. Still, the problem is not wholly different: data races can be found in the same general way, with accesses protected by guard addresses instead of lock variables. The key distinctive feature is that atomicity is explicitly associated with specific memory locations. In the case of locks, an algorithm can find out which locks protect any particular access by referring to the single, globally maintained set of currently held locks. In the case of guard statements, there is no analogous global lock set: an algorithm must keep track of whether each location is guarded or not, and by which address, individually. Our algorithms for detecting data races in programs with locks—All-Sets and Brelly (Sections 3.1 and 4.2, respectively)—can be modified to work for programs using guard statements by using an extra shadow space $H[l]$ to keep track of which lock (i.e. guard address) protects each location l instead of a global lock set H, used for accesses to all locations. Instead of updating H at each lock and unlock, the algorithms update the appropriate locations of $H[l]$ when entering and exiting guard statements, and use the appropriate $H[l]$ instead of a global H when executing its logic at a memory access.

While these modified algorithms are convenient, giving us tools for checking programs with guard statements with little extra effort, they are not optimal. Consider the bounds for All-Sets: $O(LT(k+\alpha(V,V)))$ time and $O(kLV)$ space on a computation that runs serially in time T, uses V shared memory locations, holds at most k locks simultaneously, and holds at most L different lock sets during accesses to any particular location. With guard statements, there is at most one guard address at a time per location, so $k=1$, leaving us with a time bound of $O(LT\,\alpha(V,V))$ and space bound of $O(LV)$, where L is in this context the maximum number of guard addresses used to protect any particular location. These bounds are reasonable, especially if we might expect L to be small for most programs, but they can be improved, as we will see with Review-Guards below. Now consider the bounds for Brelly: $O(kT\,\alpha(V,V))$ time and $O(kV)$ space. With $k=1$ in programs with guard statements, these bounds become $O(T\,\alpha(V,V))$ and $O(V)$—almost linear. Unfortunately, Brelly has the disadvantage of reporting violations of the umbrella discipline instead of data races directly.

We would like to have an algorithm for detecting data races in programs using guard statements that runs in nearly linear time and always reports only data races. In fact, Brelly, modified as described above, is almost such an algorithm. If locks sets always contain at most a single lock, Brelly always reports violations that are unambiguously due to data races (see lines 15–17 of the Brelly code in Figure 4-2).

---

updates are made visible to other processors upon the release of a lock.

And, when modified for guard statements, locks sets in BRELLY would contain at most a single lock (i.e. guard address) if not for the use of the fake read lock, which increases the maximum size of locks sets to two, enough to cause nonrace umbrella violations.

The solution is to check programs according to logic of the BRELLY modified for guard statements, but without the convenience of the fake read lock. Reads and writes must be handled separately, with explicit logic to ignore reads that "race" with other reads. The remaining sections of this chapter present the details of an algorithm along these lines, called REVIEW-GUARDS, and an extension of the algorithm, called REVIEW-GUARDS-SHARED, that correctly handles guard statements which contain parallelism.

6.3 The REVIEW-GUARDS algorithm

The REVIEW-GUARDS algorithm finds data races in Cilk programs with guard statements that never contain parallelism. In this section, we describe the algorithm and then prove that it acheives $O(T(\lg k + \alpha(V, V)))$ time and $O(V + k)$ space bounds on a program that, on a given input, runs serially in time T using V space, with at most k simultaneously guarded memory blocks.

During the serial depth-first execution of a program on a specific input, the REVIEW-GUARDS algorithm maintains the set of currently guarded memory blocks in the global variable *current-blocks*, which can be implemented as a red-black tree [5, Chapter 14] indexed by the start addresses of the memory blocks, allowing memory blocks to be inserted and deleted efficiently. To record information about past memory accesses, the algorithm maintains the following shadow space information for each shared memory location l: *writer*$[l]$ and *writer-guard*$[l]$, the thread ID and guard address (or NIL), respectively, of the last "serial write" to l; and *reader*$[l]$ and *reader-guard*$[l]$, the thread ID and guard address (or NIL), respectively, of the last "serial read" from l. ***Serial writes*** and ***reads***, and ***parallel writes*** and ***reads***, are analogous to BRELLY's serial and parallel accesses: they are logically in series or in parallel with the previous *writer*$[l]$ or *reader*$[l]$, respectively. These shadow spaces are initialized to NIL or the ID of the initial thread (which logically precedes all other threads) as appropriate. In addition, both *writer-guard*$[l]$ and *reader-guard*$[l]$ are tagged, when not NIL, with a *nonguarder* field indicating the thread ID of an access to l (writes for *writer-guard*$[l]$ and reads for *reader-guard*$[l]$) which is unguarded or guarded by a different address—these ***nonguarders*** are analogous to BRELLY's nonlockers. Each of these tags is associated with the guard address stored at a particular location; no tag is associated with a guard address globally, across all locations.

Besides maintaining, according to the SP-BAGS algorithm (Chapter 2), an SP-bags data structure for determining the series-parallel relationship between the current thread and any previously executed thread, REVIEW-GUARDS updates *current-blocks* appropriately whenever entering or exiting a guard statement (see Figure 6-2), and checks for data races and updates the other shadow spaces at every shared memory access (see Figure 6-3). At the begining of each memory access, the algorithm

Enter-Guard with memory blocks $p_1[n_1], p_2[n_2], \ldots$
for each memory block $p_i[n_i]$
    do insert $p_i[n_i]$ into *current-blocks*

Exit-Guard with memory blocks $p_1[n_1], p_2[n_2], \ldots$
for each memory block $p_i[n_i]$
    do delete $p_i[n_i]$ from *current-blocks*

Figure 6-2: The Review-Guards algorithm. Pseudocode for entering and exiting a guard statement. Note that all memory blocks, whether storage variables or pointers to arrays, can be considered a pointer p_i followed by the number of bytes n_i, indicating the size of the block. The global data structure *current-blocks* keeps track of the currently guarded memory blocks.

determines the guard address (if any) of the memory location being accessed by looking in *current-blocks* for a memory block containing the location (lines 1–3 of Read and Write).

Review-Guards's logic for Read and Write is closely based on Brelly's logic for Access, with writes checked against previous reads and writes and reads checked against previous writes, rather than generic accesses checked against all previous accesses. In lines 4–10 of Write, a write is checked against previous writes; in lines 17–21, a write is checked against previous reads. In lines 4–8 of Read, a read is checked against previous writes. For the benefit of future accesses, serial reads and writes update *reader*$[l]$ and *reader-guard*$[l]$, and *writer*$[l]$ and *writer-guard*$[l]$, along with the *nonguarder* tags, in lines 12–17 of Read and lines 11–16 of Write. Notice the absence of *alive* flags, which are not needed because an access races either with *reader*$[l]$, *writer*$[l]$, or a nonguarder, and so previous guard addresses that have been overwritten (or "killed," in Brelly's terminology) are irrelevant.

Review-Guards does make one addition to the basic logic of Brelly: in lines 7–8 of Write and lines 10–11 of Read, the algorithm overwrites the nonguarder of *writer-guard*$[l]$ or *reader-guard*$[l]$ with the current guard address if the current access is logically in series with the the existing nonguarder. Doing so ensures that the nonguarder will be the one logically in parallel with the most future threads.[5]

We now show that Review-Guards is correct, using an approach that is, not surprisingly, similar to the one used for Brelly in Section 4.2. The following lemma is analogous to Lemma 4.2 and follows from analogous reasoning.

Lemma 6.1 Suppose a thread e reads (or writes) some location l during an execution of Review-Guards. Then all previously executed reads (or writes) of l logically precede e in the computation. ■

[5]The reasoning here is the same as in heuristic 2 for Brelly in Section 4.4.

```
WRITE(l) in thread e
if there exists a memory block in current-blocks containing l
   then current-guard ← the start address of the block
   else current-guard ← NIL
if writer[l] ∥ e
   then if writer-guard[l] = NIL or writer-guard[l] ≠ current-guard
              then report race between writer[l] and e
                   if writer-guard[l] ≠ NIL and nonguarder[writer-guard[l]] ≺ e
                      then nonguarder[writer-guard[l]] ← e
           elseif nonguarder[writer-guard[l]] ∥ e
              then report race between nonguarder[writer-guard[l]] and current write
   else if current-guard = NIL
              then writer-guard[l] ← NIL
           elseif current-guard ≠ writer-guard[l]
              then writer-guard[l] ← current-guard
                   nonguarder[writer-guard[l]] ← writer[l]
           writer[l] ← e
if reader[l] ∥ e
   then if reader-guard[l] = NIL or reader-guard[l] ≠ current-guard
              then report race between reader[l] and current write
           elseif nonguarder[reader-guard[l]] ∥ e
              then report race between nonguarder[reader-guard[l]] and current write

READ(l) in thread e
if there exists a memory block in current-blocks containing l
   then current-guard ← the start address of the block
   else current-guard ← NIL
if writer[l] ∥ e
   then if writer-guard[l] = NIL or writer-guard[l] ≠ current-guard
              then report race between writer[l] and current read
           elseif nonguarder[writer-guard[l]] ∥ e
              then report race between nonguarder[writer-guard[l]] and current read
if reader[l] ∥ e
   then if reader-guard[l] ≠ NIL and reader-guard[l] ≠ current-guard
                and nonguarder[reader-guard[l]] ≺ e
              then nonguarder[reader-guard[l]] ← e
   else if current-guard = NIL
              then reader-guard[l] ← NIL
           elseif current-guard ≠ reader-guard[l]
              then reader-guard[l] ← current-guard
                   nonguarder[reader-guard[l]] ← reader[l]
           reader[l] ← e
```

Figure 6-3: The REVIEW-GUARDS algorithm. Pseudocode for accessing shared memory locations.

Throughout the proofs of the following lemma and theorem we use the fact that at all times the value in *reader-guard*$[l]$ (*writer-guard*$[l]$) corresponds to the access recorded in *reader*$[l]$ (*writer*$[l]$)—i.e. it is the guard address of that access or NIL if the access was unguarded. This correspondence holds because *reader-guard*$[l]$ and *reader*$[l]$ (*writer-guard*$[l]$ and *writer*$[l]$) are always updated together in lines 12–17 of READ (lines 11–16 of WRITE). We also rely on the fact that, during any access to l, *current-guard*$[l]$ is the guard address of l, or NIL if none; this is ensured by the logic of ENTER-GUARD and EXIT-GUARD.

Lemma 6.2 Suppose *reader-guard*$[l] \neq$ NIL at some point during the execution of REVIEW-GUARDS, and let e be the most recent thread performing a serial read from l without being protected by the guard address in *reader-guard*$[l]$, or the initial thread if no such serial read exists. Then *nonguarder*[*reader-guard*$[l]$] is either e or a thread after e in the serial execution in series with it. The analogous statement holds for *writer-guard*$[l]$ and *nonguarder*[*writer-guard*$[l]$].

Proof: Fix a moment in the execution and let e be as defined in the lemma. We first show that *nonguarder*[*reader-guard*$[l]$] is set to e at some point and then it is always updated, if at all, to a thread after it in the serial execution in series with it. All line numbers refer to READ(l).

Let e' be the first serial read after e in the serial execution; such a serial read exists because *reader-guard*$[l] \neq$ NIL and *reader-guard*$[l]$ is only set to a guard address during a serial read in line 15. Since e is the most recent serial read from l not protected by *reader-guard*$[l]$, e' is protected by *reader-guard*$[l]$ and therefore updated *nonguarder*[*reader-guard*$[l]$] to the previous *reader*$[l]$, namely e, in line 16. (Recall that *reader*$[l]$ is initialized to the ID of the initial thread.)

Now consider, in order of the serial execution, each thread e'' that updates *non-guarder*[*reader-guard*$[l]$] after e', if any such thread exists. Now e'' must be a parallel read, since every serial read from l after e is protected by *reader-guard*$[l]$, causing the test in line 14 to fail and preventing line 16 to be run. Thus, *nonguarder*[*reader-guard*$[l]$] is updated by e'' to e'' (the most recent thread at the time) after the algorithm checks that e'' is in series with the previous value of *nonguarder*[*reader-guard*$[l]$] (lines 10–11). By transitivity of $\prec$, we know the whatever thread ends up in *nonguarder*[*reader-guard*$[l]$], if not e, runs after e and is in series with it.

The analogous statement about *writer-guard*$[l]$ and *nonguarder*[*writer-guard*$[l]$] follows from analogous reasoning. ∎

Theorem 6.3 Consider a Cilk program with guard statements that never contain parallelism. The REVIEW-GUARDS algorithm detects a data race in the computation of this program running serially on a given input if and only if a data race exists in the computation.

Proof: ($\Rightarrow$) We now show that data races reported indeed exist. Given that at least one write is involved, REVIEW-GUARDS reports a race in the following two cases:

Case 1: The current access is in parallel with *reader* (*writer*) and either *reader-guard* (*writer-guard*) is NIL or a different guard address than the one protecting the current access, so a race between the current access and *reader* (*writer*) is reported—e.g. line 6 of WRITE. In this case, the existence of a data race as reported follows because *reader-guard* (*writer-guard*) always corresponds to the guard address of *reader* (*writer*).

Case 2: The current access uses the same guard as *reader-guard* (*writer-guard*) but is in parallel with *nonguarder*[*reader-guard*] (*nonguarder*[*writer-guard*]), so a race between the current access and the nonguarder is reported—e.g. line 10 of WRITE. In this case, the existence of a data race as reported follows from Lemma 6.2.

($\Leftarrow$) Suppose a data race exists; we show that the algorithm reports a race. Let e_1 and e_2 be threads which form a data race on l, where e_2 executes after e_1 in the serial execution. Since there is a race, the threads did not use the same guard address while accessing l or at least one of the accesses was unguarded.

Suppose e_1 reads and e_2 writes, and that $reader[l] = e$ when e_2 runs. If $e = e_1$, then consider what happens in WRITE(l) when e_2 runs: the algorithm will discover that $reader[l] \parallel e_2$ (since $reader[l] = e = e_1$) and that $reader\text{-}guard[l]$ is e_1's guard address or NIL, causing a race to be reported in line 19. We now assume that $e \neq e_1$ and consider two cases depending on whether e_1 is a serial or parallel read. Note that since $reader[l]$ and $reader\text{-}guard[l]$ correspond, $reader\text{-}guard[l]$ will be the guard address used by e, or NIL if none, at the time e_2 runs.

Case 1: The access in e_1 is a serial read, so it updates $reader[l] \leftarrow e_1$. In this case, e must run after e_1 since $reader[l] = e$ when e_2 runs. By Lemma 6.1 we have $e_1 \prec e$; and then by $e_1 \parallel e_2$ and Lemma 2.1 we have $e \parallel e_2$. Consider what happens in WRITE(l) when e_2 runs. If $reader\text{-}guard[l]$ is NIL or different from e_2's current guard address (*current-guard*), then a race will be reported in line 19.

Otherwise, $current\text{-}guard[l] = reader\text{-}guard[l]$ and we need to show that $nonguarder[reader\text{-}guard[\mathrm{l}]] \parallel e_2$ and that a race is reported in line 21. Consider the most recent serial read e' before e which was unguarded or protected by a different guard address than $reader\text{-}guard[l]$ (e's guard address). Since the serial read in e_1 was either unguarded or not protected by $current\text{-}guard[l] = reader\text{-}guard[l]$ (there would be no race with e_2 otherwise), either $e' = e_1$ or e' ran after e_1, in which case by Lemma 6.1 we have $e_1 \prec e'$. By Lemma 6.2 we know that $nonguarder[reader\text{-}guard[l]]$ either equals e' or is a thread after e' in series with it, so either trivially or by transitivity we have $e_1 \prec nonguarder[reader\text{-}guard[\mathrm{l}]]$. By Lemma 2.1 we then have $nonguarder[reader\text{-}guard[l]] \parallel e_2$, as desired.

Case 2: The access in e_1 is a parallel read, and so does not update $reader[l]$. Let e' be the most recent thread which updates $reader[l]$ before e_1. Then $e' \parallel e_1$, since otherwise e_1 would have updated $reader[l]$ in line 17 of READ(l). By pseudotransitivity (Lemma 2.2) we have $e' \parallel e_2$. Either $e' = e$ and thus $e \parallel e_2$ or, by Lemma 6.1, $e' \prec e$, which further means, by Lemma 2.1, that $e \parallel e_2$. Consider what happens in WRITE(l) when e_2 runs. As in the previous case above, if *reader-guard*[l] is NIL or different from *current-guard*[l], a race will be reported in line 19.

Otherwise *reader-guard*[l] = *current-guard*[l], and we need to show that *nonguarder*[*reader-guard*[l]] will be discovered to be in parallel with e_2, leading to a race being reported in line 21. Let e'' be the most recent serial read not protected by *reader-guard*[l] (e's guard address), or the initial thread if no such read exists.

Suppose e'' either equals e' or that it ran after e', in which case by Lemma 6.1 we have $e' \prec e''$. By Lemma 6.2 we have either *nonguarder*[*reader-guard*[l]] = e'' or that *nonguarder*[*reader-guard*[l]] is a thread after e'' in series with it. Thus, either trivially or by applying transitivity (to get $e' \prec$ *nonguarder*[*reader-guard*[l]]) and Lemma 2.1 (using the fact that $e' \parallel e_2$), we have *nonguarder*[*reader-guard*[l]] $\parallel e_2$, as desired.

Now suppose e'' ran before e', and let w be the thread stored in *nonguarder*[*reader-guard*[l]] at the time the read in e_1 executed. Since e'' is the most recent serial read not protected by e's guard address (the same as e_2's) or no such read exists, at the time e_1 runs (which is after e') we have *reader-guard*[l] $\neq$ NIL and *reader-guard*[l] $\neq$ *current-guard* (since e', being after e'', has the same guard address as e_2 but e_1 does not). Thus, since e_1 is a parallel read, it checked whether $w \prec e_1$, and if it was, set *nonguarder*[*reader-guard*[l]] $\leftarrow e_1$ (lines 10–11 of READ(l)). As any serial read after e'' used the same guard address as e and so *nonguarder*[*reader-guard*[l]] is not updated in a serial read, any Subsequent updates to *nonguarder*[*reader-guard*[l]] are in parallel reads and therefore only update *nonguarder*[*reader-guard*[l]] with serial values (lines 10–11). By transitivity we have $e_1 \prec$ *nonguarder*[*reader-guard*[l]] and by Lemma 2.1 we have *nonguarder*[*reader-guard*[l]] $\parallel e_2$. If, however, the read in e_1 found that w was in parallel in line 10 of READ(l) and so did not update *nonguarder*[*reader-guard*[l]], then by pseudotransitivity (Lemma 2.2) we have $w \parallel e_2$, and by similar reasoning we can by transitivity conclude that $w \prec$ *nonguarder*[*reader-guard*[l]] and thus, by Lemma 2.1, that *nonguarder*[*reader-guard*[l]] $\parallel e_2$, as desired.

The argument when e_1 is a write and e_2 is either a read or a write is analogous. ■

In proving the runtime performance of REVIEW-GUARDS, we assume that *current-blocks* is maintained as a red-black tree indexed by the start addresses of the memory blocks. Insertion and deletion of memory blocks in ENTER-GUARD and EXIT-GUARD are handled in the obvious way. In lines 1–3 of READ and WRITE, the algorithm finds the guard address (if any) of a location by searching the tree for either a memory block that starts with the location, in which case the location itself is the guard address, or the predecessor to the memory location in the tree, in which case the predecessor's start address is the guard address *if* that memory block contains the location. Since insert, delete, search, and search-for-predecessor operations for a red-black tree run in logarithmic time, and the size of *current-blocks* is at most k, the operations involving *current-blocks* each run in $O(\lg k)$ time.

Theorem 6.4 Consider a Cilk program with guard statements that never contain parallelism which runs serially on a given input in T, uses V shared memory locations, and guards at most k memory blocks simultaneously. The REVIEW-GUARDS algorithm checks this computation in $O(T(\lg k+\alpha(V,V)))$ time using $O(V+k)$ space.

Proof: The space bound follows since a constant amount of shadow space information per shared memory location, and *current-blocks* takes $O(k)$ space. The time bound follows since, in addition to the red-black tree operations during ENTER-GUARD, EXIT-GUARD, and each memory access, the algorithm performs, at each access, a constant number of series/parallel relationships checks, each taking $O(\alpha(V,V))$ time. ∎

Since k is almost always at most 2 or 3, it is probably not worth using a red-black tree in practice. Instead, the *current-blocks* data structure can be kept as an unsorted linked list, with operations on it taking $O(k)$ time.

6.4 The REVIEW-GUARDS-SHARED algorithm

In this section, we present the REVIEW-GUARDS-SHARED algorithm, an extension of REVIEW-GUARDS that allows for parallelism within critical section. The logic of REVIEW-GUARDS-SHARED is, oddly enough, almost exactly a concatenation of the logic of REVIEW-GUARDS with that of SP-BAGS, the algorithm upon which all the algorithms in thesis thesis, including REVIEW-GUARDS, are based. After describing REVIEW-GUARDS-SHARED, we show that it is correct and prove that it has the same, almost-linear performance bounds as the original REVIEW-GUARDS.

As discussed in Section 3.2, we require that all procedures spawned in a critical section always finish before the end of the critical section. Therefore, just as we require a sync after the last spawn in a critical section ended by a `Cilk_unlock` statement, we extend the semantics of guard statements to include an implicit sync just before the exit from a guard statement.

When guard statements contain parallelism, two parallel accesses protected by the same guard address may still form a data race, since they may share the same "instance" of the guard address. Given a guarded access to location l, the ***instance*** of the guard address of that access is the outermost guard statement in the runtime computation which protects a memory block including l—in other words, the current guard statement that actually acquired a lock, or otherwise provides atomicity, for l's guard address. If two parallel accesses to the same location, at least one of which is a write, share the same instance of a guard address, the accesses form a data race.

Parallelism within guard statements is considerably easier to handle than parallelism within locked critical regions, because there is always at most one guard

ENTER-GUARD with memory blocks $p_1[n_1], p_2[n_2], \ldots$
1 **for** each memory block $p_i[n_i]$
2   **do** insert $p_i[n_i]$ into *current-blocks*
3   **for** each location l in $p_i[n_i]$
4   **do** *inside-reader*$[l] \leftarrow$ ID of initial thread
5   *inside-writer*$[l] \leftarrow$ ID of initial thread

EXIT-GUARD with guard statement ID s and memory blocks $p_1[n_1], p_2[n_2], \ldots$
1 perform EXIT-GUARD logic from the original REVIEW-GUARDS

WRITE(l) in thread e
1 perform WRITE(l) logic from the original REVIEW-GUARDS
2 **if** *current-guard* $\neq$ NIL $\triangleright$ *current-guard* variable from line 1
3   **then if** *inside-writer*$[l] \parallel e$
4   **then** declare race between *inside-writer*$[l]$ and current write
5   **if** *inside-reader*$[l] \parallel e$
6   **then** declare race between *insider-reader*$[l]$ and current write
7   *inside-writer*$[l] \leftarrow e$

READ(l) in thread e
1 perform READ(l) logic from the original REVIEW-GUARDS
2 **if** *current-guard* $\neq$ NIL $\triangleright$ *current-guard* variable from line 1
3   **then if** *inside-writer*$[l] \parallel e$
4   **then** declare race between *inside-writer*$[l]$ and current read
5   **if** *inside-reader*$[l] \prec e$
6   **then** *inside-reader*$[l] \leftarrow e$

Figure 6-4: The REVIEW-GUARDS-SHARED algorithm. Logic is shown for for entering and exiting a guard statement, and for reading and writing a shared memory location.

address protecting any given location. To find data races involving parallelism within guard statements, a detection algorithm need only check a guarded access against other accesses sharing the instance of its one guard address. In fact, it would be sufficient to find basic determinacy races on a location l within guard statements that protect location l, since any accesses to l within such guard statements cannot also be guarded by any other guard statements.

Accordingly, REVIEW-GUARDS-SHARED extends REVIEW-GUARDS by simply performing the logic of SP-BAGS, the efficient determinacy race detection algorithm (Chapter 2), to find races on a location l within each guard statement that protects l. SP-BAGS maintains $reader[l]$ and $writer[l]$ at each location l to record information about past reads and writes to l in the SP-tree; REVIEW-GUARDS-SHARED maintains, if l is currently guarded, $inside\text{-}reader[l]$ and $inside\text{-}writer[l]$ to do the same thing, except only for the part of the SP-tree corresponding to the code inside the guard statement that protects l. At the entry to a guard statement that protects l, REVIEW-GUARDS-SHARED resets $inside\text{-}reader[l]$ and $inside\text{-}writer[l]$ to the value they would have at the beginning of SP-BAGS (the ID of the initial thread), so that the determinacy-race detection logic can begin anew for the part of the computation within that guard statement.

The logic for REVIEW-GUARDS-SHARED is shown in Figure 6-4. Lines 3–6 of READ and lines 3–7 of WRITE exactly mirror the read and write logic in SP-BAGS (see [9]). The logic for entering and exiting guard statements is exactly as in the original REVIEW-GUARDS algorithm (Figure 6-2), except that in lines 4–5 of ENTER-GUARD, the $inside\text{-}reader[l]$ and $insider\text{-}writer[l]$ fields are reset to the ID of the initial thread.

Theorem 6.5 The REVIEW-GUARDS-SHARED algorithm detects a data race in the computation of a Cilk program with guard statements running serially on a given input if and only if a data race exists, assuming that any parallelism within guard statements is restricted as described above.

Proof: That REVIEW-GUARDS-SHARED correctly finds data races between unguarded accesses or accesses protected by different guard addresses follows trivially from the correctness of REVIEW-GUARDS (Theorem 6.3). That it correctly finds data races between guarded accesses that share the same instance of a guard address follows from the correctness SP-BAGS (see [9]), and the fact that REVIEW-GUARDS-SHARED performs SP-BAGS's logic for a location l within each guard statement that protects l. ∎

Theorem 6.6 Consider a Cilk program with guard statements that runs serially on a given input in T time using V shared memory locations. The REVIEW-GUARDS-SHARED algorithm checks this computation in $O(T(\lg k+\alpha(V,V)))$ time using $O(V+k)$ space.

Proof: Follows trivially from the performance bounds of the REVIEW-GUARDS and SP-BAGS algorithms. ∎

Chapter 7

Conclusion

After summarizing this thesis, we conclude by discussing two questions. First, we outline several issues related to the question, "How should atomicity be specified?" Second, we consider the pitfalls of detecting apparent rather than feasible races, explaining how the correctness guarantees proven for the algorithms in this thesis, though of limited practical use, are suggestive of debugging methodologies involving the cooperation between program annotation and algorithms with guarantees.

Summary of thesis

This thesis has presented three algorithms for data-race detection in multithreaded programs, as well as extensions to these algorithms for handling critical sections containing parallelism. We have constrained the debugging problem by considering only the computation of a program running serially on a given input, which in Cilk corresponds to a left-to-right depth-first treewalk of the series-parallel parse tree representing the program's dynamically unfolding dag of threads. Threads have been considered to be "logically in parallel" according to this parse tree, taking no account of program semantics which may cause nonexisting races to show up as apparent races in the tree. Given these limiting assumptions, we have shown that our algorithms are guaranteed to find data races (or umbrella discipline violations) in computations if and only if any exist.

The first algorithm, ALL-SETS, detects (apparent) data races precisely, with slowdown and space-blowup factors (i.e. the factors by which a computation's running time is slowed down and space usage increased) dominated by L, the maximum number of lock sets held during accesses to any particular location. Even though L may grow with input size for some applications, likely causing the algorithm to be impractical, ALL-SETS is the fastest data-race detection algorithm seen to date.

The second algorithm, BRELLY, debugs computations asymptotically faster than ALL-SETS, with slowdown and space-blowup factors of only k, the maximum number of simultaneously held locks. BRELLY achieves this gain in efficiency at the cost of flexibility and precision: rather than detecting data races directly, it detects (apparent) violations of the "umbrella locking discipline," which precludes some race-free locking protocols as well as data races. Preliminary testing shows that BRELLY is indeed significantly faster than All-Sets in practice when L is not a small constant. We also have preliminary experience showing that the number of nonrace violations reported by Brelly can be significant, due to apparent but infeasible violations caused by the practice of "memory publishing," described below. Despite its limitations, the umbrella discipline in more flexible than other locking disciplines proposed in the context of data race detection. We have also presented several heuristics that can conservatively determine whether an umbrella violation is caused by a data race.

The third algorithm, Review-Guards, checks for (apparent) data races in the serial computations of programs using a proposed "guard statement," rather than locks, to specify atomicity. This algorithm takes advantage of the higher-level semantics of guard statements, which associate atomic operations with specific memory locations, to achieve nearly linear performance in both time and space. We know of no other algorithm for detecting data races in the context of a language construct similar to our guard statement.

We have given -Shared extensions for each of these three algorithm which correctly handle critical sections containing parallelism. The extensions for All-Sets and Brelly perform only a factor of k worse than the originals in time and space, while the extension of Review-Guards performs with the same nearly-linear asymptotic performance as the original.

In the course of presenting these algorithms, we have given a useful model for programs with critical sections containing parallelism, and, in extending our basic algorithms, have given examples of how to use this model in a way that may generalize to other algorithms. We have also specified the guard statement and suggested how it might be implemented, given the stipulation of "same-start semantics," which is motivated by runtime efficiency.

What should atomicity look like?

A theme of our work in this thesis has been the question: how should atomicity be specified? A good answer to this question must satisfy concerns at several different levels. Atomicity should be expressed in a way that is easy for programmers to use and understand. Many different kinds of applications, with various atomic operations and data structures, need to be provided for. The overhead of atomicity during runtime must be minimal. And, of course, there should be efficient and reliable algorithms for detecting data races in programs containing atomicity. The following are several issues that arise in the search for good tradeoffs between these often competing interests.

The "right" locking discipline. Is the umbrella discipline in any meaningful way the right locking discipline? We have seen that adopting it allows significant and provable performance gains in debugging, but does it correspond to any useful programming methodology? Since the discipline allows different parallel subcomputations that are in series with each other to use different locks for each location, we expect the main utility of the discipline to be in modular software development. If, within each parallel module, a programmer ensures that one lock (or set of locks) always protects each particular data structure, the modules can be combined in series without unifying their locking schemes.

Other disciplines are either easier to understand or may be more flexible, however. Enforcing a single lock that protects each location throughout an entire program is straightforward, and it is not overly restrictive for the many programs that naturally follow such a rule. One possibly useful discipline that is more flexible than the umbrella discipline is the requirement that each "writer-centric parallel region" (instead of each umbrella) use a single lock to protect each location. A writer-centric parallel region consists of all the accesses to some location that are logically in parallel with some write to that location. Since umbrellas encompass both writer- and reader-centric parallel regions (defined analogously), enforcing fixed locks per location only in writer-centric regions is more flexible. It may also be more natural, since a write always races with parallel accesses while a read in parallel with another read is never a race. We have investigated algorithms for checking for violations of a discipline based on writer-center regions but found nothing promising.[1]

Are guard statements too limiting? We have no experience using the guard statement in real-world applications. We do expect that its syntax and semantics are sufficient for many programs, in which the needed atomic operations fit naturally with the built-in structure, and which do not need more than same-start semantics. And as we have seen with the parallel table example at the end of Section 6.1, programming tasks for which guard statements, as given, are not sufficient may be intrinsically difficult, even with the full flexibility of user-level locking.

Efficient implementation of the original semantics of guard statements. Guard statements would be easier to think about and more flexible to use if we could do away with the requirement that guarded blocks of memory start with the same address. To do this, we need to find a runtime efficient implementation of the original semantics, in which atomicity is truly considered per-location, with no reference to blocks of memory and start addresses. As for data-race detection, Review-Guards can easily be modified to work with the original semantics: in Enter-Guard, update the current guard address of each location to itself rather than to the start address of the block of memory which contains it. We consider it unlikely, however, that an efficient implementation exists: again, there seem to be intrinsic difficulties with performing atomic operations on overlapping blocks of memory.

Guard statements with locks. Perhaps an ideal programming language would provide both guard statements and locks to users, giving them to the option of using either or both in any particular situation. Are there general guidelines for programs

[1] The question of the "right" locking discipline can also be asked with reference of deadlock detection. The problem of detecting deadlocks, even "apparent" deadlocks in a computation dag, is NP-hard [28]. We have investigated using locking disciplines based on the notion of umbrellas to make the problem of deadlock detection easier, with no progress. Is there some useful locking discipline which precludes deadlock and for which violations are easy (or just tractable) to detect?

using guard statements and locks or unexpected complications arising from their interaction, perhaps relating to deadlock? As for data-race detection, such "mixed" programs could be checked by a further variation of the algorithms for locks modified for guard statements, described in Section 6.2. The algorithm keeps *both* a global lock set H and per-location guard addresses $H[l]$. Then, each access to l is considered to be protected (as it indeed is) by $H \cup H[l]$, with the rest of the algorithm as usual.

Since $H[l]$ contains at most a single guard address per location l, the maximum size of the unioned lock set would be one more than the maximum number of locks held simultaneously. Thus, the asymptotic bounds for the variants of All-Sets and Brelly for mixed programs would be the same as for the original algorithms for locks alone. To handle mixed programs with critical sections (of either kind) containing parallelism, the logic for Review-Guards-Shared could easily be merged, also without degrading asymptotic performance, into the -Shared versions of the algorithms for locks.

Apparent versus feasible data races

The algorithms we presented in this thesis are guaranteed to find all apparent data races (or discipline violations) in a computation. The utility of this guarantee is significantly weakened by the existence of apparent but infeasible races, which lead to false alarms. Apparent races are races between accesses that appear to be in parallel according to the SP-tree that captures the semantics of the parallel control commands in a program's execution—i.e. the accesses are *logically* in parallel. Apparent races may or may not actually be feasible since alternate control flows due to changes in scheduling may cause certain accesses to never happen at all: so they either happen in one order (after some other access, for instance) or not at all. Our algorithms cannot distinguish between apparent and feasible races because they rely solely on the structure of spawns and syncs in the computation to deduce the series/parallel relationships between threads.

In our experience, a significant source of apparent but infeasible data races is the practice of "memory publishing." Suppose some thread allocates some memory from global storage, writes to the memory, and then "publishes" the memory by making a pointer to it available to other threads through a global, shared data structure. Then, if another thread logically in parallel with this thread reads from the newly published memory, the two threads will form an apparent data race—even though the read from the memory could only happen after the write, since the memory was not globally accessible until after the memory was published after the write. Apparent data races due to memory publishing is common in programs operating on global linked-list and other dynamic data structures in parallel. Stark gives a thorough discussion of the problem in [26].

Stark also presents in [26] a theory of nondeterminism, in which one feature is the notion of an "abelian" program. Intuitively, a program is abelian if its critical section commute—i.e. they produce the same results regardless of the order in which they execute. Stark shows that abelian programs without data races (and without deadlocks) produce the same computation when running on a given input no matter how threads are scheduled. Further, he shows that if a feasible data race exists in an abelian program, an apparent race will exist in any computation of that program. Thus, the algorithms presented in this thesis can be used to certify that an abelian program, when running on a given input, either produces determinate results or contains a data race.

What about nonabelian programs? The guarantee that the algorithms in this thesis find all data races (or discipline violations) in a computation seem to be rather pointless for these programs. Indeed, for nonabelian programs, the guarantees are not in themselves particularly useful, except perhaps as an vague indication that our algorithms may tend to catch more data races than algorithms with no guarantees.

But the chief value of our guarantees—the if-and-only-if correctness proofs of our algorithms—is that they are suggestive. We know we cannot expect program verification: finding feasible data races exactly is NP-hard [20]. But there may be a *via media* between intractable problems and merely heuristic solutions. For instance, could programmers annotate code to help race detection algorithms "understand" the program's semantics better? The practice of memory publishing, which can cause many apparent but infeasible data races in a computation, might be alleviated, if not decidedly solved, in this way. Suppose a programmer annotated code that publishes memory in a way that a compiler could tell a debugging tool what was happening. Perhaps the debugging tool could then figure out when apparent data races were caused by the publication of memory. A simpler idea is to have a programmer use fake locks to "protect" the creation and later access of data object after publication against each other. (These explicit fake locks are akin to the hidden fake read lock used by the All-Sets and Brelly algorithms; see Chapter 2.) In essence, the programmer uses a fake lock to tell the debugging algorithm that the two operations do not form a data race. He translates application-specific logic into something simple that detection algorithms can understand: locks. There may very well be other ways in which program annotation and algorithms, with guarantees, can work together.

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] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. Detecting data races in Cilk programs that use locks. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1998. To appear.

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

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

[6] 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.

[7] 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.

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

[9] 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.

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

[11] 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.

[12] 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.

[13] John L. Hennessy and David A. Patterson. *Computer Architecture: a Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, second edition, 1998.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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`.

[19] 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.

[20] Robert H. B. Netzer and Barton P. Miller. On the complexity of event ordering for shared-memory parallel program executions. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages II: 93–97, August 1990.

[21] 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.

[22] 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.

[23] 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.

[24] Keith H. Randall. *Cilk: Efficient Multithreaded Computing.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1998.

[25] 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.

[26] Andrew F. Stark. Debugging multithreaded programs that incorporate user-level locks. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1998.

[27] Robert Endre Tarjan. Applications of path compression on balanced trees. *Journal of the Association for Computing Machinery*, 26(4):690–715, October 1979.

[28] Mihalis Yannakakis. Freedom from deadlock of safe locking policies. *SIAM Journal on Computing*, 11(2):391–408, May 1982.