Detecting Data Races in Cilk Programs that Use Locks

Guang-Ien Cheng* Mingdong Feng† Charles E. Leiserson* Keith H. Randall* Andrew F. Stark*

Abstract

When two parallel threads holding no locks in common access the same memory location and at least one of the threads modifies the location, a "data race" occurs, which is usually a bug. This paper describes the algorithms and strategies used by a debugging tool, called the Nondeterminator-2, which checks for data races in programs coded in the Cilk multithreaded language. Like its predecessor, the Nondeterminator, which checks for simple "determinacy" races, the Nondeterminator-2 is a debugging tool, not a verifier, since it checks for data races only in the computation generated by a serial execution of the program on a given input.

We give an algorithm, ALL-SETS, that determines whether the computation generated by a serial execution of a Cilk program on a given input contains a race. For a program that runs serially in time T, accesses V shared memory locations, uses a total of n locks, and holds at most $k \ll n$ locks simultaneously, ALL-SETS runs in $O(n^k T\,\alpha(V,V))$ time and $O(n^k V)$ space, where α is Tarjan's functional inverse of Ackermann's function.

Since ALL-SETS may be too inefficient in the worst case, we propose a much more efficient algorithm which can be used to detect races in programs that obey the "umbrella" locking discipline, a programming methodology that is more flexible than similar disciplines proposed in the literature. We present an algorithm, BRELLY, which detects violations of the umbrella discipline in $O(kT\,\alpha(V,V))$ time using $O(kV)$ space.

We also prove that any "abelian" Cilk program, one whose critical sections commute, produces a determinate final state if it is deadlock free and if it generates any computation which is data-race free. Thus, the Nondeterminator-2's two algorithms can verify the determinacy of a deadlock-free abelian program running on a given input.

Keywords

Algorithm, Cilk, data race, debugging, multithreading, parallel programming, race detection.

*MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, Massachusetts 02139, USA.

†Department of Information Systems and Computer Science, National University of Singapore, 10 Lower Kent Ridge Road, Republic of Singapore 119260.

This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Grants N00014-94-1-0985 and F30602-97-1-0270.

To appear in *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Puerto Vallarta, June 1998.

```
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: A Cilk program with a data race. Cilk [3, 4, 6, 15, 20] is a multithreaded parallel language based on C being developed at the MIT Laboratory for Computer Science. The `spawn` statement in a Cilk program creates a parallel subprocedure, and the `sync` statement provides control synchronization to ensure that all spawned subprocedures have completed. The function `Cilk_lock()` acquires a specified lock, and `Cilk_unlock()` releases a currently held lock.

1 Introduction

In a parallel multithreaded computation, a ***data race*** exists if logically parallel threads access the same location, the two threads hold no locks in common, and at least one of the threads writes to the location. A data race is usually a bug, because depending on how the threads are scheduled, the program may exhibit unexpected, nondeterministic behavior. If the two threads hold a lock in common, however, the nondeterminism is not usually a bug. By introducing locks, the programmer presumably intends to allow the locked critical sections to be scheduled in either order, as long as they are not interleaved.

Figure 1 illustrates a data race in a Cilk program. The procedures `foo1`, `foo2`, and `foo3` run in parallel, resulting in parallel accesses to the shared variable `x`. The accesses by `foo1` and `foo2` are protected by lock `A` and hence do not form a data race. Likewise, the accesses by `foo1` and `foo3` are protected by lock `B`. The accesses by `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 statements in `foo2` and `foo3` are composed of multiple machine instructions which may interleave, possibly resulting in a lost update to `x`.

Since a data race is usually a bug, automatic data-race detection has been studied extensively. Static race detectors [25] can sometimes determine whether a program will ever produce a data race when run on all possible inputs. Since static debuggers cannot fully understand the semantics of programs, however, most race detectors are dynamic tools in which potential races are detected at runtime by executing the program on a given input. Some dynamic race detectors perform a post-mortem analysis based on program execution traces [12, 18, 23, 26], while others perform an "on-the-fly" analysis during program execution. On-the-fly debuggers directly instrument memory accesses via the compiler [9, 10, 13, 14, 22, 29], by binary rewriting [32], or by augmenting the machine's cache coherence protocol [24, 30].

The race-detection algorithms in this paper are based on the Nondeterminator [13], which finds "determinacy races" in Cilk programs that do not use locks. The Nondeterminator executes a Cilk program serially on a given input, maintaining an efficient "SP-bags" data structure to keep track of the logical series/parallel relationships between threads. For a Cilk program that runs serially in time T and accesses V shared-memory locations, the Nondeterminator runs in $O(T\,\alpha(V,V))$ time and $O(V)$ space, where α is Tarjan's functional inverse of Ackermann's function, which for all practical purposes is at most 4.

The Nondeterminator-2, which is currently under development, finds data races in Cilk programs that use locks. This race detector contains two algorithms, both of which use the same efficient SP-bags data structure from the original Nondeterminator. The first of these algorithms, ALL-SETS, is an on-the-fly algorithm which, like most other race-detection algorithms, assumes that no locks are held across parallel control statements, such as `spawn` and `sync`. The second algorithm, BRELLY, is a faster on-the-fly algorithm, but in addition to reporting data races as bugs, it also reports as bugs some complex (but race-free) locking protocols.

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

In previous work, Dinning and Schonberg's "lock-covers" algorithm [10] also detects all data races in a computation. The ALL-SETS algorithm improves the lock-covers algorithm by generalizing the data structures and techniques from the original Nondeterminator to produce better time and space bounds. Perkovic and Keleher [30] offer an on-the-fly race-detection algorithm that "piggybacks" on a cache-coherence protocol for lazy release consistency. Their approach is fast (about twice the serial work, and the tool runs in parallel), but it only catches races that actually occur during a parallel execution, not those that are logically present in the computation.

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

A ***locking discipline*** is a programming methodology that dictates a restriction on the use of locks. For example, many programs adopt the discipline of acquiring locks in a fixed order so as to avoid deadlock [19]. Similarly, the "umbrella" locking discipline precludes data races. It requires that each location be protected by the same lock within every parallel subcomputation of the computation. Threads that are in series may use different locks for the same location (or possibly even none, if no parallel accesses occur), but if two threads in series are both in parallel with a third and all access the same location, then all three threads must agree on a single lock for that location. If a program obeys the umbrella discipline, a data race cannot occur, because parallel accesses are always protected by the same lock. The BRELLY algorithm detects violations of the umbrella locking discipline.

Savage et al. [32] 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 relationship of 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, and importantly, supports a more flexible discipline. In particular, the umbrella discipline allows separate program modules to be composed in series without agreement on a global lock for each location. For example, an application may have three phases—an initialization phase, a work phase, and a clean-up phase—which can be developed independently without agreeing globally on the locks used to protect locations. If a fourth module runs in parallel with all of these phases and accesses the same memory locations, however, the umbrella discipline does require that all phases agree on the lock for each shared location. Thus, although the umbrella discipline is more flexible than Eraser's discipline, it is more restrictive than what a general data-race detection algorithm, such as ALL-SETS, permits.

Most dynamic race detectors, like ALL-SETS and BRELLY, attempt to find, in the terminology of Netzer and Miller [28], ***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. Thus, an apparent data race between two threads in a given computation may not actually be feasible, because the computation itself may change if the threads were scheduled in a different order. Since the problem of exactly finding feasible data races is computationally difficult,[1] attention has naturally focused on the easier (but still difficult) problem of finding apparent data races.

[1]Even in simple models, finding feasible data races is NP-hard [27].

For some classes of programs, however, a feasible data race on a given input exists if and only if an apparent data race exists in every computation for that input. To check for a feasible data race in such a program, it suffices to check a single computation for an apparent data race. One class of programs having this property are "abelian" programs in which critical sections protected by the same lock "commute": intuitively, they produce the same effect regardless of scheduling. For a computation generated by a deadlock-free abelian program running on a given input, we prove that if no data races exist in that computation, then the program is ***determinate***: all schedulings produce the same final result. For abelian programs, therefore, ALL-SETS and BRELLY can verify the determinacy of the program on a given input. Our results on abelian programs formalize and generalize the claims of Dinning and Schonberg [10, 11], who argue that for "internally deterministic" programs, checking a single computation suffices to detect all races in the program.

The remainder of this paper is organized as follows. Section 2 presents the ALL-SETS algorithm, and Section 3 presents the BRELLY algorithm. Section 4 gives some empirical results obtained by using the Nondeterminator-2 in its ALL-SETS and BRELLY modes. Section 5 defines the notion of abelian programs and proves that data-race free abelian programs produce determinate results. Section 6 offers some concluding remarks.

2 The All-Sets Algorithm

In this section, we present the ALL-SETS algorithm, which detects data races in Cilk computations that use locks. We first give some background on Cilk and the series-parallel control structure of its computations. We then discuss locking in Cilk. Finally, we present the ALL-SETS algorithm itself, show that it is correct, and analyze its performance.

The computation of a Cilk program on a given input can be viewed as a directed acyclic graph, or ***dag***, in which vertices are instructions and edges denote ordering constraints imposed by control statements. A Cilk `spawn` statement generates a vertex with out-degree 2, and a Cilk `sync` statement generates a vertex whose in-degree is 1 plus the number of subprocedures syncing at that point. Normal execution of serial code results in a linear chain of vertices, which we call a ***thread***. A thread cannot contain parallel control statements.

The computation dag generated by a Cilk program can itself be represented as a binary ***series-parallel parse tree***, as illustrated in Figure 2 for the program in Figure 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 relation between two threads e_1 and e_2 to be determined by examining their least common ancestor, which we denote by $\text{LCA}(e_1, e_2)$. If $\text{LCA}(e_1, e_2)$ is a P-node, the two threads are logically in parallel, which we denote by $e_1 \parallel e_2$. If $\text{LCA}(e_1, e_2)$ 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.

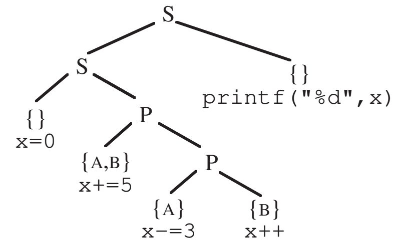


Figure 2: The series-parallel parse tree for the Cilk program in Figure 1, abbreviated to show only the accesses to shared location `x`. Each leaf is labeled with a code fragment that accesses `x`, with the lock set for that access shown above the code fragment.

Release 5.1 of Cilk [6] 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. 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`. We assume in this paper, as does the general literature, that any lock/unlock pair is contained in a single thread, and thus holding a lock across a parallel control construct is forbidden.[2] The ***lock set*** of an access is the set of locks held by the thread when the access occurs. The ***lock set*** of several accesses is the intersection of their respective lock sets.

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

The ALL-SETS algorithm is based on the efficient SP-BAGS algorithm used by the original Nondeterminator to detect determinacy races in Cilk programs that do not use locks. The SP-BAGS algorithm executes a Cilk program on a given input in serial, depth-first order. This execution order mirrors that of normal C programs: every subcomputation that is spawned executes completely before the procedure that spawned it continues. While executing the program, SP-BAGS maintains an SP-bags data structure based on Tarjan's nearly linear-time least-common-ancestors algorithm [33]. The SP-

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

ACCESS(l) in thread e with lock set H

1 **for** each $\langle e', H' \rangle \in lockers[l]$
2   **do if** $e' \parallel e$ and $H' \cap H = \{\}$
3   **then** declare a data race
4 $redundant \leftarrow$ FALSE
5 **for** each $\langle e', H' \rangle \in lockers[l]$
6   **do if** $e' \prec e$ and $H' \supseteq H$
7   **then** $lockers[l] \leftarrow lockers[l] - \{\langle e', H' \rangle\}$
8   **if** $e' \parallel e$ and $H' \subseteq H$
9   **then** $redundant \leftarrow$ TRUE
10 **if** $redundant =$ FALSE
11   **then** $lockers[l] \leftarrow lockers[l] \cup \{\langle e, H \rangle\}$

Figure 3: The ALL-SETS algorithm. The operations for the `spawn`, `sync`, and `return` actions 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 add and remove locks from the lock set H appropriately.

bags data structure allows SP-BAGS to determine the series/parallel relation 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 to determine previous threads that have accessed the same location as the current thread. 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.

The ALL-SETS algorithm also uses the SP-bags data structure 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 was 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_unlock(&A);
Cilk_lock(&B); Cilk_lock(&C);
WRITE(l)
Cilk_unlock(&C); Cilk_unlock(&B);
```

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

The ALL-SETS algorithm is shown in Figure 3. Intuitively, this algorithm records all lockers, but it is careful to prune redundant lockers, keeping at most one locker per distinct lock set. Lines 1–3 check to see if a data race has occurred and report any violations. Lines 5–11 then add the current locker to the *lockers* shadow space and prune redundant lockers.

Before proving the correctness of ALL-SETS, we restate two important lemmas from [13].

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

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

We now prove that the ALL-SETS algorithm is correct.

Theorem 3 *The* ALL-SETS *algorithm detects a data race in a computation of a Cilk program running 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 $H_1 \cap H_2 = \{\}$.

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

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

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

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

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

Proof: First, observe that no two lockers in *lockers* have the same lock set, because the logic in lines 5–11 ensure that if $H = H'$, then locker $\langle e, H\rangle$ either replaces $\langle e', H'\rangle$ (line 7) or is considered redundant (line 9). Thus, there are at most L lockers in the list $lockers[l]$. Each lock set takes at most $O(k)$ space, so the space needed for *lockers* is $O(kLV)$. The length of the list $lockers[l]$ determines the number of series/parallel relations that are tested. In the worst case, we need to perform $2L$ such tests (lines 2 and 6) and $2L$ set operations (lines 2, 6, and 8) per access. Each series/parallel test takes amortized $O(\alpha(V,V))$ time, and each set operation takes $O(k)$ time. Therefore, the ALL-SETS algorithm runs in $O(TL(k+\alpha(V,V)))$ time. ∎

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

3 The Brelly Algorithm

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. In this section, we formally define the umbrella discipline and present the BRELLY algorithm for detecting violations of this discipline. We prove that the BRELLY algorithm is correct and analyze its performance, which we show to be asymptotically better than that of ALL-SETS.

The umbrella discipline can be defined precisely in terms of the parse tree of a given Cilk computation. An ***umbrella*** of accesses to a location l is a subtree rooted at a P-node containing accesses to l in both its left and right subtrees, as is illustrated in Figure 4. An umbrella of accesses to l is ***protected*** if its accesses have a nonempty lock set and ***unprotected*** otherwise. A program obeys the ***umbrella locking discipline*** if it contains no unprotected umbrellas. In other words, within each umbrella of accesses to a location l, all threads must agree on at least one lock to protect their accesses to l.

The next theorem shows that adherence to the umbrella discipline precludes data races from occuring.

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

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

The umbrella discipline can also be violated by unusual, but data-race free, locking protocols. For instance, suppose that a location

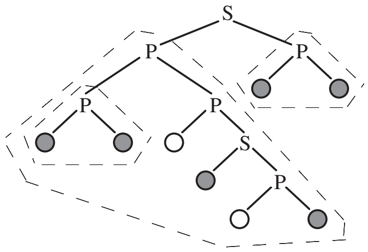


Figure 4: Three umbrellas of accesses to a location l. In this parse tree, each shaded leaf represents a thread that accesses l. Each umbrella of accesses to l is enclosed by a dashed line.

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

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 Section 2). Figure 5 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. Like the ALL-SETS algorithm, BRELLY also maintains a set H of currently held locks. In addition, BRELLY maintains two shadow spaces of shared memory: *accessor*, which stores for each location the thread that performed the last "serial access" to that location; and *locks*, which stores the lock set of that access. Each entry in the *accessor* space is initialized to the initial thread (which logically precedes all threads in the computation), and each entry in the *locks* space is initialized to the empty set.

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

Whenever BRELLY encounters an access by a thread e to a location l, it checks for a violation with previous accesses to l, updating the shadow spaces appropriately for future reference. If $accessor[l] \prec e$, we say the access is a ***serial access***, and the algorithm performs lines 2–5, setting $locks[l] \leftarrow H$ and $accessor[l] \leftarrow e$, as well as updating $nonlocker[h]$ and $alive[h]$ appropriately for each $h \in H$. If $accessor[l] \parallel e$, we say the access is a ***parallel access***, and the algorithm performs lines 6–17, killing the locks in $locks[l]$

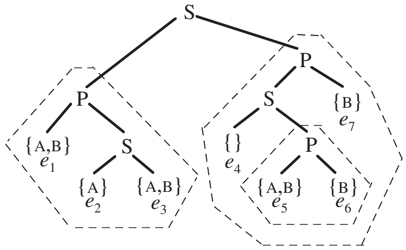
```
ACCESS(l) in thread e with lock set H
1   if accessor[l] ≺ e
2     then ▷ serial access
            locks[l] ← H, leaving nonlocker[h] with its old
              nonlocker if it was already in locks[l] but
              setting nonlocker[h] ← accessor[l] otherwise
3           for each lock h ∈ locks[l]
4               do alive[h] ← TRUE
5           accessor[l] ← e
6     else ▷ parallel access
7           for each lock h ∈ locks[l] − H
8               do if alive[h] = TRUE
9                     then alive[h] ← FALSE
10                         nonlocker[h] ← e
11          for each lock h ∈ locks[l] ∩ H
12              do if alive[h] = TRUE and nonlocker[h] || e
13                    then alive[h] ← FALSE
14          if no locks in locks[l] are alive (or locks[l] = {})
15            then report violation on l involving
                      e and accessor[l]
16                 for each lock h ∈ H ∩ locks[l]
17                     do report access to l without h
                              by nonlocker[h]
```

Figure 5: 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 [13].

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 6 illustrates how BRELLY works. The umbrella containing threads e_1, e_2, and e_3 is protected by lock A but not by lock B, which is reflected in *locks*$[l]$ after thread e_3 executes. The umbrella containing e_5 and e_6 is protected by B but not by A, which is reflected in *locks*$[l]$ after thread e_6 executes. During the execution of thread e_6, A is killed and *nonlocker*[A] is set to e_6, according to the logic in lines 7–10. When e_7 executes, B remains as the only lock alive in *locks*$[l]$ and *nonlocker*[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.



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 6: A sample execution of the BRELLY algorithm. We restrict our attention to the algorithm's operation on a single location l. In the parse tree, each leaf represents an access to l and is labeled with the thread that performs the access (e.g., e_1) and the lock set of that access (e.g., $\{\mathrm{A}, \mathrm{B}\}$). 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.

The following two lemmas, which will be helpful in proving the correctness of BRELLY, are stated without proof.

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

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

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

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

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

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

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

In the second case, we shall show that w, the thread stored in $nonlocker[h]$ when h is killed, is a descendant of p, which implies that $h \notin U$, because by Lemma 7, w accesses l without the lock h. Assume for the purpose of contradiction that w is not a descendant of p. Then, we have $\text{LCA}(w, e_0) = \text{LCA}(w, e')$, 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 6, we therefore have the contradiction that $w \prec e_0$. If line 2 sets $nonlocker[h] \leftarrow w$, then w performs a serial access, which must be prior to the most recent serial access by e_0. By Lemma 6, we once again obtain the contradiction that $w \prec e_0$.

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

We first show that at this point, $accessor[l]$ is a left-descendant of p. Assume for the purpose of contradiction that $accessor[l]$ is not a left-descendant of p (and is therefore not a descendant of p at all), and let $p' = \text{LCA}(accessor[l], p)$. 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), $e' \parallel e_{left}$ 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. ∎

Theorem 9 *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 *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 5 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 Experimental Results

We are in the process of implementing both the ALL-SETS and BRELLY algorithms as part of the Nondeterminator-2 debugging tool. Our experiences are therefore highly preliminary. In this section, we describe our initial results from running these two algorithms on four Cilk programs that use locks. Our implementations of ALL-SETS and BRELLY have not yet been optimized, and so better performance than what we report here is likely to be possible.

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

`maxflow`: A maximum-flow code based on Goldberg's push-relabel method [16]. Each vertex in the graph contains a lock. Parallel threads perform simple operations asynchronously on graph edges and vertices. To operate on a vertex u, a thread acquires u's lock, and to operate on an edge (u, v), the thread acquires both u's

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

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

Figure 7: 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.

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 [7, 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 7 shows the preliminary results of our experiments on the test codes. These results indicate that the performance of ALL-SETS is indeed dependent on the parameter L. Essentially no performance difference exists between ALL-SETS and BRELLY when $L = 1$, but ALL-SETS gets progressively worse as L increases. On all of our test programs, BRELLY runs fast enough to be useful as a debugging tool. In some cases, ALL-SETS is as fast, but in other cases, the overhead of ALL-SETS is too extreme (iteration 13 of `rad` takes over 3.5 hours) to allow interactive debugging.

5 Abelian Programs

By checking a single computation for the absence of determinacy races, the original Nondeterminator can guarantee that a Cilk program without locking is determinate: it always produces the same answer (when run on the same input). To date, no similar claim has been made by any data-race detector for programs with locks. We cannot make a general claim either, but in this section, we introduce a class of nondeterministic programs for which a determinacy claim can be made. We prove that the absence of data races in a single computation of a deadlock-free "abelian" program implies that the program (when run on the same input) is determinate. As a consequence, ALL-SETS and BRELLY can verify the determinacy of abelian programs from examining a single computation. We do not claim that abelian programs form an important class in any practical sense. Rather, we find it remarkable that a guarantee of determinacy can be made for any nontrivial class of nondeterministic programs.

```
int x, y;
Cilk_lockvar A;

cilk int main() {
  Cilk_lock_init(&A);
  x = 0;
  spawn bar1();
  spawn bar2();
  sync;
  printf("%d", y);
}

cilk void bar1() {
  Cilk_lock(&A);
  x++;
  if (x == 1)
     y = 3;
  Cilk_unlock(&A);
}

cilk void bar2() {
  Cilk_lock(&A);
  x++;
  Cilk_unlock(&A);
  y = 4;
}
```

Figure 8: A Cilk program that generates a computation with an infeasible data race on the variable `y`.

Locking introduces nondeterminism intentionally, allowing many different computations to arise from the same program, some of which may have data races and some of which may not. Since ALL-SETS and BRELLY examine only one computation, they cannot detect data races that appear in other computations. More subtlely, the data races that these algorithms do detect might actually be infeasible, never occurring in an actual program execution.

Figure 8 shows a program that exhibits an infeasible data race. In the computation generated when `bar1` obtains lock `A` before `bar2`, a data race exists between the two updates to `y`. In the scheduling where `bar2` obtains lock `A` first, however, `bar1`'s update to `y` never occurs. In other words, no scheduling exists in which the two updates to `y` happen simultaneously, and in fact, the final value of `y` is always 4. Thus, the computation generated by the serial depth-first scheduling, which is the one examined by ALL-SETS and BRELLY, contains an infeasible data race.

Deducing from a single computation that the program in Figure 8 is determinate appears difficult. But not all programs are so hard to understand. For example, the program from Figure 1 exhibits a race no matter how it is scheduled, and therefore, ALL-SETS and BRELLY can always find a race. Moreover, if all accesses to `x` in the program were protected by the same lock, no data races would exist in any computation. For such a program, checking a single computation for the absence of races suffices to guarantee that the program is determinate. The reason we can verify the determinacy of this program from a single computation is because it has "commuting" critical sections.

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

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

In practice, therefore, most programs are not abelian, but abelian programs nevertheless form a nontrivial class of nondeterministic programs that can be checked for determinacy. For example, all programs that use locking to accumulate values atomically, such as a histogram program, fall into this class. Although abelian programs form an arguably small class in practice, the guarantees of determinacy that ALL-SETS and BRELLY provide for them are not provided by any other existing race-detectors for *any* class of lock-employing programs. It is an open question whether a more general class of nondeterministic programs exists for which an efficient race-detector can offer a provable guarantee of determinacy.

In order to study the determinacy of abelian programs, we first give a formal multithreaded machine model that more precisely describes an actual execution of a Cilk program. We view the abstract execution machine for Cilk as a (sequentially consistent [21]) shared memory together with a collection of ***interpreters***, each with some private state. (See [5, 8, 17] for examples of multithreaded implementations similar to this model.) Interpreters are dynamically created during execution by each `spawn` statement. The ith such child of an interpreter is given a unique ***interpreter name*** by appending i to its parent's name.

When an instruction is ***executed*** by an interpreter, it maps the current state of the multithreaded machine to a new state. An interpreter whose next instruction cannot be executed is said to be ***blocked***. If all interpreters are blocked, the machine is ***deadlocked***.

Although a multithreaded execution may proceed in parallel, we consider a serialization of the execution in which only one interpreter executes at a time, but the instructions of the different interpreters may be interleaved.[4] The initial state of the machine consists of a single interpreter whose program counter points to the first instruction of the program. At each step, a nondeterministic choice among the current nonblocked interpreters is made, and the instruction pointed to by its program counter is executed. The resulting sequence of instructions is referred to as an ***execution*** of the program.

When an instruction executes in a run of a program, it affects the state of the machine in a particular way. To formalize the effect of an instruction execution, we define an ***instantiation*** of an instruction to be a 3-tuple consisting of an instruction I, the shared memory location l on which I operates (if any), and the name of the interpreter that executes I. We assume that the instantiation of an instruction is a deterministic function of the machine state.

We define a ***region*** to be either a single instantiation other than a LOCK or UNLOCK instruction, or a sequence of instantiations that comprise a critical section (including the LOCK and UNLOCK instantiations themselves).[5] Every instantiation belongs to at least one region and may belong to many. Since a region is a sequence of instantiations, it is determined by a particular execution of the program and not by the program code alone. We define the ***nesting count*** of a region R to be the maximum number of locks that are acquired in R and held simultaneously at some point in R.

The execution of a program can alternatively be viewed as sequence of instantiations, rather than instructions, and an instantiation sequence can always be generated from an instruction sequence. We formally define a ***computation*** as a dag in which the vertices are instantiations and the edges denote synchronization. Edges go from each instantiation to the next instantiation executed by the same interpreter, from each spawn instantiation to the first instantiation executed by the spawned interpreter, and from the last instantiation of each interpreter to the next sync instantiation executed by its parent interpreter.

We can now give a more precise definition of a data race. A ***data race*** exists in a computation if two logically parallel instantiations access the same memory location without holding the same lock, and at least one of the accesses is a WRITE. Since a memory location is a component of each instantiation, it is unambiguous what it means for two instantiations to access the same memory location. In contrast, if the computation were constructed so that the nodes were instructions, it would not be apparent from the dag alone whether two nodes reference the same memory location.

A ***scheduling*** of a computation G is a sequence of instantiations forming a permutation of the vertex set of G. This sequence must satisfy the ordering constraints of the dag, as well as have the property that any two LOCK instantiations that acquire the same lock are separated by an UNLOCK of that lock in between. If any scheduling of any prefix of G can be extended to a scheduling of G, we say that G is ***deadlock free***. Not every scheduling of G corresponds to some actual execution of the program. If a scheduling does correspond to an actual execution as defined by the machine model, we call that scheduling a ***true scheduling*** of G; otherwise it is a ***false scheduling***. Since we are only concerned with the final memory states of true schedulings, we define two schedulings of G to be ***equivalent*** if both are false, or both are true and have the same final memory state. An alternate definition of commutativity, then, is that two regions R_1 and R_2 commute if, beginning with any reachable machine state S, the instantiation sequences R_1R_2 and R_2R_1 are equivalent.

Our study of the determinacy of abelian programs will proceed as follows. Starting with a data-race free, deadlock-free computa-

[4]The fact that any parallel execution can be simulated in this fashion is a consequence of our choice of sequential consistency as the memory model.

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

tion G resulting from the execution of an abelian program, we first prove that adjacent regions in a scheduling of G can be commuted. Second, we show that regions which are spread out in a scheduling of G can be grouped together. Third, we prove that all schedulings of G are true and yield the same final memory state. Finally, we prove that all executions of the abelian program generate the same computation and hence the same final memory state.

Lemma 10 (Reordering) *Let G be a data-race free, deadlock-free computation resulting from the execution of an abelian program. Let X be some scheduling of G. If regions R_1 and R_2 appear adjacent in X, i.e., $X = X_1R_1R_2X_2$, and $R_1 \parallel R_2$, then the two schedulings $X_1R_1R_2X_2$ and $X_1R_2R_1X_2$ are equivalent.*

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

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

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

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

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

Case 2: Lock A does not appear in R_2. We start with the sequence $X_1R_1R_2X_2$ and commute pieces of R_1 one at a time with R_2: first, the instantiation UNLOCK(A), then the (immediate) subregions of R_1, and finally the instantiation LOCK(A). The instantiations LOCK(A) and UNLOCK(A) commute with R_2, because A does not appear anywhere in R_2. Each subregion of R_1 commutes with R_2 by the inductive hypothesis, because each subregion has lower nesting count than R_1. After commuting all of R_1 past R_2, we have an equivalent execution $X_1R_2R_1X_2$.

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

Lemma 11 (Region grouping) *Let G be a data-race free, deadlock-free computation resulting from the execution of an abelian program. Let X be some scheduling of G. Then, there exists an equivalent scheduling X' of G in which the instantiations of every region are contiguous.*

Proof: We shall create X' by grouping the regions in X one at a time. Each grouping operation will not destroy the grouping of already grouped regions, so eventually all regions will be grouped.

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

Consider the partially overlapping region S with the leftmost UNLOCK in R. Then all subregions of S which have any instantiations inside R are completely inside R and are therefore contiguous. We remove S by moving each of its (immediate) subregions in R to just left of R using commuting operations. Let S_1 be the leftmost subregion of S which is also in R. We can commute S_1 with every instruction I to its left until it is just past the start of R. There are three cases for the type of instruction I. If I is not a LOCK or UNLOCK, it commutes with S_1 by Lemma 10 because it is a region in parallel with S_1. If $I =$ LOCK(B) for some lock B, then S_1 commutes with I, because S_1 cannot contain LOCK(B) or UNLOCK(B). If $I =$ UNLOCK(B), then there must exist a matching LOCK(B) inside R, because S is chosen to be the region with the leftmost UNLOCK without a matching LOCK. Since there is a matching LOCK in R, the region defined by the LOCK/UNLOCK pair must be contiguous by the choice of R. Therefore, we can commute S_1 with this whole region at once using Lemma 10.

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

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

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

Lemma 12 *Let G be a data-race free, deadlock-free computation resulting from the execution of an abelian program. Then every scheduling of G is true and yields the same final memory state.*

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

We construct this set using Lemma 11. Let X' and Y' be the schedulings of G with contiguous regions which are obtained by applying Lemma 11 to X and Y, respectively. From X' and Y', we can commute whole regions using Lemma 10 to put their threads in the serial depth-first order specified by G, obtaining schedulings X'' and Y''. We have $X'' = Y''$, because a computation has only one serial depth-first scheduling. Thus, all schedulings X, X', $X'' = Y''$, Y', and Y are equivalent. Since X is a true scheduling, so is Y, and both have the same final memory state. ■

Theorem 13 *An abelian Cilk program that produces a deadlock-free computation with no data races is determinate.*

Proof: Let X be an execution of an abelian program that generates a data-race free, deadlock-free computation G. Let Y be an arbitrary execution of the same program. Let H be the computation generated by Y, and let H_i be the prefix of H that is generated by the first i instantiations of Y. If H_i is a prefix of G for all i, then $H = G$, and therefore, by Lemma 12, executions X and Y have the same final memory state. Otherwise, assume for contradiction that i_0 is the largest value of i for which H_i is a prefix of G. Suppose that the (i_0+1)st instantiation of Y is executed by an interpreter with name η. We shall derive a contradiction through the creation of a new scheduling Z of G. We construct Z by starting with the first i_0 instantiations of Y, and next adding the successor of H_{i_0} in G that is executed by interpreter η. We then complete Z by adding, one by one, any nonblocked instantiation from the remaining portion of G. One such instantiation always exists because G is deadlock free. By Lemma 12, the scheduling Z that results is a true scheduling of G. We thus have two true schedulings which are identical in the first i_0 instantiations but which differ in the (i_0+1)st instantiation. In both schedulings the (i_0+1)st instantiation is executed by interpreter η. But, the state of the machine is the same in both Y and Z after the first i_0 instantiations, which means that the (i_0+1)st instantiation must be the same for both, which is a contradiction. ■

We state without proof one more lemma, which allows us to show that ALL-SETS and BRELLY can give a guarantee of determinacy for deadlock-free abelian programs.

Lemma 14 *Let G be a computation generated by a deadlock-free abelian program. If G is data-race free, then it is deadlock free.* ■

Corollary 15 *If the* ALL-SETS *algorithm detects no data races in an execution of a deadlock-free abelian Cilk program, then the program running on the same input is determinate.*

Proof: Combine Theorems 3 and 13 and Lemma 14. ■

Corollary 16 *If the* BRELLY *algorithm detects no violations of the umbrella discipline in an execution of a deadlock-free abelian Cilk program, then the program run on the same input is determinate.*

Proof: Combine Theorems 5, 8, and 13 and Lemma 14. ■

6 Conclusion

Although ALL-SETS and BRELLY are fast race-detection algorithms, many practical questions remain as to how to use these algorithms to debug real programs. In this section, we discuss our early experiences in using the Nondeterminator-2, which currently provides both algorithms as options, to debug Cilk programs.

A key decision by Cilk programmers is whether to adopt the umbrella locking discipline. A programmer might first debug with ALL-SETS, but unless he has adopted the umbrella discipline, he will be unable to fall back on BRELLY if ALL-SETS seems too slow. We recommend that programmers use the umbrella discipline initially, which is good programming practice in any event, and only use ALL-SETS if they are forced to drop the discipline.

The Nondeterminator-2 reports any apparent data race as a bug. As we have seen, however, some data races are infeasible. We have experimented with ways that the user can inform the Nondeterminator-2 that certain races are infeasible, so that the debugger can avoid reporting them. One approach we have tried is to allow the user to "turn off" the Nondeterminator-2 in certain pieces of code using compiler pragmas and other linguistic mechanisms. Unfortunately, turning off the Nondeterminator-2 requires the user to check for data races manually between the ignored accesses and all other accesses in the program. A better strategy has been to give the user fake locks—locks that are acquired and released only in debugging mode, as in the implicit R-LOCK fake lock. The user can then protect accesses involved in apparent but infeasible races using a common fake lock. Fake locks reduce the number of false reports made by the Nondeterminator-2, and they require the user to manually check for data races only between critical sections locked by the same fake lock.

Another cause of false reports is "publishing." One thread allocates a heap object, initializes it, and then "publishes" it by atomically making a field in a global data structure point to the new object so that the object is now available to other threads. If a logically parallel thread now accesses the object in parallel through the global data structure, an apparent data race occurs between the initialization of the object and the access after it was published. Fake locks do not seem to help much, because it is hard for the initializer to know all the other threads that may later access the object, and we do not wish to suppress data races among those later accesses. We do not yet have a good solution for this problem.

With the BRELLY algorithm, some programs may generate many violations of the umbrella discipline that are not caused by actual data races. We have implemented several heuristics in the Nondeterminator-2's BRELLY mode to report straightforward data races and hide violations that are not real data races whenever possible.

False reports are not a problem when the program being debugged is abelian, but programmers would like to know whether an ostensibly abelian program is actually abelian. Dinning and Schonberg give a conservative compile-time algorithm to check if a program is "internally deterministic" [10], and we have given thought to how the abelian property might likewise be conservatively checked. The parallelizing compiler techniques of Rinard and Diniz [31] may be applicable.

We are currently investigating versions of ALL-SETS and BRELLY that correctly detect races even when parallelism is allowed within critical sections. A more ambitious goal is to detect potential deadlocks by dynamically detecting the user's accordance with a flexible locking discipline that precludes deadlocks.

Acknowledgments

Thanks to Arvind of MIT for helpful discussions about locking. Thanks to the other members of the Cilk group for their support. We are also indebted to the SPAA reviewers for their extensive comments on the submitted draft.

References

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

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

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

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

[5] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

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

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

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

[9] 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.

[10] 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.

[11] Anne Carolyn Dinning. *Detecting Nondeterminism in Shared Memory Parallel Programs*. PhD thesis, Department of Computer Science, New York University, July 1990.

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

[13] 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.

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

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

[16] 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.

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

[18] 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.

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

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

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

[22] 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.

[23] 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.

[24] 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.

[25] 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`.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

[32] Stefan Savage, Michael Burrows, Greg Nelson, Patric 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.

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