Efficiently Detecting Races in Cilk Programs That Use Reducer Hyperobjects

I-Ting Angelina Lee
Washington University in St. Louis
One Brookings Drive
St. Louis, MO 63130

Tao B. Schardl
MIT CSAIL
32 Vassar Street
Cambridge, MA 02139

ABSTRACT

A multithreaded Cilk program that is ostensibly deterministic may nevertheless behave nondeterministically due to programming errors in the code. For a Cilk program that uses reducers, a general reduction mechanism supported in various Cilk dialects, such programming errors are especially challenging to debug, because the errors can expose the nondeterminism in how the Cilk runtime system manages a reducer.

We identify two unique types of races that arise from incorrect use of reducers in a Cilk program and present two algorithms to catch them. The first algorithm, called the Peer-Set algorithm, detects view-read races, which occur when the program attempts to retrieve a value out of a reducer when the read may result a nondeterministic value, such as before all previously spawned subcomputations that might update the reducer have necessarily returned. The second algorithm, called the SP+ algorithm, detects determinacy races, instances where a write to a memory location occurs logically in parallel with another access to that location, even when the raced-on memory locations relate to reducers. Both algorithms are provably correct, asymptotically efficient, and can be implemented efficiently in practice. We have implemented both algorithms in our prototype race detector, Rader. When running Peer-Set, Rader incurs a geometric-mean multiplicative overhead of 2.32 over running the benchmark without instrumentation. When running SP+, Rader incurs a geometric-mean multiplicative overhead of 16.76.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*; D.2.5 [**Software Engineering**]: Testing and Debugging; D.3.3 [**Programming Languages**]: Language Constructs and Features—*concurrent programming structures*

Keywords

Cilk; determinacy race; nondeterminism; reducers; view-read race

This research was supported in part by NSF Grant 1314547. Tao B. Schardl was supported in part by an MIT Akamai Fellowship and an NSF Graduate Research Fellowship.

SPAA'15, June 13–15, 2015, Portland, OR, USA.

DOI: http://dx.doi.org/10.1145/2755573.2755599.

1. INTRODUCTION

A multithreaded Cilk program that is "ostensibly deterministic" may nevertheless behave nondeterministically due to programming errors in the code. Typically these errors, also called ***races***, occur when the program fails to coordinate parallel operations on a shared variable, causing accesses and updates to be performed on the variable in a nondeterministic order based on scheduling happenstance. Although provably efficient and correct race detection algorithms exist for Cilk computations[1] [3, 15, 23], they do not provide the same guarantees when the program under test employs a "reducer hyperobject" [18], an advanced linguistic feature supported in various Cilk dialects. Races involving the use of a reducer are particularly challenging to debug, because such races can expose the nondeterminism in how the Cilk runtime system manages a reducer. This paper addresses the question of how to efficiently and correctly detect races in Cilk programs that use reducers.

Many modern concurrency platforms provide some form of ***reduction mechanism*** [18, 22, 25, 28, 34, 39, 42, 45] to support safe parallel updates to shared variables. A reduction mechanism coordinates parallel updates to a shared variable by applying the parallel updates to distinct ***views*** of the variable. When the parallel subcomputations that update the variable complete, these views are combined together, or ***reduced***, using a binary ***reduce operator***. A reduction mechanism typically encapsulates the nondeterministic behavior induced by parallel updates as long as the update and reduce operations satisfy associativity and commutativity.

Reducer hyperobjects (or ***reducers*** for short) [18], which are supported by Cilk dialects including Intel Cilk Plus [22], Cilk++ [26], and Cilk-M [24], provide a general reduction mechanism for Cilk programs and exhibit several useful properties.

- Reducers operate on arbitrary Cilk code. They are not tied to any particular linguistic construct.
- Reducers can operate on any abstract data type, including a set, a linked list, or even a *user-defined data type*, so long as the user supplies an appropriate reduce operator.
- To produce a deterministic result, a reducer's update and reduce operations do not need to be commutative; associativity suffices.

In contrast, other reduction mechanisms, such as OpenMP's reduction clause [34] or Microsoft's PPL's combinable objects [28], tie the reduction mechanism to a particular construct, such as a parallel loop, or require reductions to be commutative.

Although these properties make reducers a powerful general-purpose reduction mechanism, they leave open opportunities for programming errors that can produce races involving reducers. Such errors can, in particular, expose the nondeterminism in the

[1] Henceforth, when we say a Cilk program, we mean a Cilk program with a specific given input. On the other hand, when we say a Cilk computation, we mean an instance of an execution of a Cilk program with a given input.

Cilk runtime system's efficient management of reducers, which includes two significant optimizations [18]. First, a new reducer view is created only when a ***worker*** thread ***steals*** some parallel subcomputation. Second, views are reduced together in an opportunistic fashion, causing reductions to occur in a nondeterministic order. Consequently, the state of a reducer's view at a particular program point, the number of views created throughout the execution, and when the views are reduced together are all nondeterministic, depending on how the scheduling plays out. This nondeterminism is typically encapsulated by the reducers when used and programmed correctly, but it can become observable due to programming errors.

The incorrect use of a reducer gives rise to two unique types of races. The first type of race, called a ***view-read race***, occurs when a Cilk computation reads the value of a reducer at a program point where the read might produce a nondeterministic value, such as before all previously spawned subcomputations that might update the reducer have necessarily returned. Because the Cilk runtime system creates and reduces views based on scheduling, such a read can cause multiple runs of the same Cilk program to produce different results. A second type of race, called a ***determinacy race*** (also called a ***general race*** [31]), occurs when two logically parallel instructions operate on the same memory location, and at least one of them is a write. Although ordinary Cilk programs can contain determinacy races, a Cilk program that uses a reducer can contain a determinacy race involving an ***view-aware*** instruction executed in updating or reducing views of a reducer. (In contrast, we refer to all other instructions that do not operate on views as ***view oblivious***.) Such a determinacy race is particularly challenging to debug, because a view-aware instruction involved in a race might not execute at all if the Cilk runtime system schedules the computation differently and thus manages the views differently.

Existing algorithms for detecting determinacy races in Cilk computations, including the SP-bags algorithm [15], the SP-order algorithm [3], and the SP-hybrid algorithm [3], do not support detecting races involving reducers. Extending these race detection algorithms to handle reducers while providing provable guarantees is non-trivial for two reasons. First, the use a reducer generates parallel control dependencies that violate the structural assumptions that these algorithms depend on. Specifically, the computation can no longer be modeled as a "series-parallel dag" [15], which is a property that existing algorithms rely on. Second, different runs of a Cilk program that uses a reducer can cause different view-aware instructions to be executed, depending how the scheduling plays out. Providing complete coverage could potentially require executing exponentially many different schedules to elicit all possible view-aware instructions. Consequently, existing tools that embody the SP-bags algorithm,[2] such as the Nondeterminator [15] and Cilk Screen [23], cannot guarantee correctness when one of the instructions involved in a race is executed to operate on a reducer view.

Contributions

In this paper, we show how to efficiently and correctly detect these two types of races in a Cilk computation that uses reducers. Specifically, we make the following contributions.

The Peer-Set algorithm. We present the Peer-Set algorithm, which executes a Cilk computation serially and analyzes its logical parallelism to detect view-read races. The algorithm is provably correct, meaning it reports a view-read race if and only if the Cilk computation contains one. For a Cilk computation that runs in time T on one processor, the Peer-Set algorithm executes in time $O(T\alpha(v,v))$, where α is Tarjan's functional inverse of Ackermann's function, a very slowly growing function which, for all practical purposes, is bounded above by 4.

The SP+ algorithm. We present the SP+ algorithm, which detects determinacy races in Cilk computations that use reducers. The SP+ algorithm extends Feng and Leiserson's SP-bags algorithm [15] for detecting determinacy races in ordinary Cilk programs that do not employ reducers. The SP+ algorithm takes as input a Cilk program, its input, and a ***steal specification*** that effectively fixes the schedule. That is, a steal specification specifies the program points at which steals occur and which reduce operations execute. Like the Peer-Set algorithm, SP+ executes the computation serially albeit simulates the steals according to the steal specification to detect determinacy races. The SP+ algorithm is provably correct, meaning it reports a determinacy race in the computation if and only if one exists, regardless of whether that determinacy race occurs due to an operation on a reducer. Furthermore, the SP+ algorithm executes efficiently in time $O((T + M\tau)\alpha(v,v))$, where T is the running time of the Cilk program on the given input on 1 processor, M is the number of steals in the steal specification, and τ is the worst-case running time of a reduce operation. The SP+ algorithm thus incurs overhead over the SP-bags algorithm only to execute reduce operations and simulate necessary steals.

Implementation and empirical evaluation of the algorithms. We have developed a prototype tool, called ***Rader***, that implements both the Peer-Set and SP+ algorithms to debug Cilk computations that use reducers. Rader implements the Peer-Set and SP+ algorithms by using compiler instrumentation to track memory accesses and parallel control dependencies. Using Rader, we empirically demonstrate the efficiency of both algorithms in practice. We ran Rader on 6 application benchmarks that use reducers. Compared to running each benchmark without instrumentation, Rader incurred geometric-mean multiplicative overheads of 2.32 and 16.76 to run the Peer-Set and SP+ algorithms, respectively.

Analysis of SP+'s coverage guarantees. We show how the SP+ algorithm can be used to efficiently check all executions of an "ostensibly deterministic" Cilk program for determinacy races that involve at least one view-oblivious instruction. A single run of the SP+ algorithm detects determinacy races in one possible schedule and thus has limited coverage; it elicits only a subset of all possible view-aware instructions. Although an exponential number of steal specifications exist for a given Cilk program, one can do better for most Cilk programs. Most Cilk programs are written to be ***ostensibly deterministic***, meaning that, in the absence of a race, its view-oblivious instructions are fixed across all executions regardless of scheduling, and that it employs only reducers with semantically associative reduce operations. For such Cilk programs, we show how to construct a polynomial number of steal specifications to elicit all possible view-aware instructions. The SP+ algorithm can use these steal specifications to exhaustively check for determinacy races between view-oblivious and view-aware instructions.

The remainder of the paper is organized as follows. Section 2 discusses the relevant background and provides an example of a program that contains races involving reducers. Sections 3 and 4 present the Peer-Set algorithm and the intuition for its correctness. Sections 5 and 6 present the SP+ algorithm and some intuition for its correctness. Section 7 shows that executing SP+ with polynomial number of different steal specifications is necessary and sufficient to elicit all possible view-aware instructions in a ostensibly deterministic Cilk program, thereby providing the stated coverage guarantees. Section 8 describes our prototype implementation of Rader and empirically evaluates its performance. Section 9 discusses related work and Section 10 provides concluding remarks.

[2] To the best of our knowledge, no implementation of the SP-order and SP-hybrid algorithms exists.

2. EXAMPLES OF RACES THAT INVOLVE A REDUCER

This section provides an motivational example to illustrate how races that involve operations on a reducer can occur. We review Cilk linguistics and semantics, including that of reducer hyperobjects. We walk through the example to illustrate how a subtle programming error can trigger a race between user code and a reduce operation on a reducer.

Cilk-style dynamic multithreading. Cilk extends C/C++ with the keywords `cilk_spawn`, `cilk_sync`, and `cilk_for`, of which `cilk_spawn` and `cilk_sync` are more primitive. Parallelism is created using the keyword `cilk_spawn`. When a function f invokes another function g by preceding the invocation with `cilk_spawn`, g is ***spawned***, and the scheduler may continue to execute the ***continuation*** of f — the statement after the spawning of g — in parallel with g, without waiting for g to return. The complement of `cilk_spawn` is `cilk_sync`, which acts as a local barrier and joins together, or ***sync***, the parallelism specified by `cilk_spawn`. When a function f reaches a `cilk_sync`, the Cilk runtime ensures that control in f does not pass the `cilk_sync` until all functions spawned previously in f have completed and returned. The `cilk_for` keyword defines a parallel loop — all loop iterations may run in parallel with each other — which may be understood in terms of `cilk_spawn` and `cilk_sync`.

Note that these keywords denote the *logical parallelism* of the computation, rather than the actual parallel execution. During execution, Cilk's work-stealing scheduler [5,19] dynamically load balances a parallel computation across available ***worker*** threads while respecting the dependencies specified by these keywords. Typically, a worker executes a Cilk computation in its ***serial order*** — at a `cilk_spawn`, the worker executes the spawned function before its continuation. When a worker runs out of work, it becomes a ***thief*** and chooses a ***victim*** worker at random to ***steal*** from. If the victim worker has excess work, the thief may steal some of this work by resuming the continuation of some function. Notably, a worker's behavior mirrors precisely the behavior of a serial execution between successful steals, and Cilk's support for reducers implements significant optimizations based on this fact.

Cilk reducer hyperobjects. A reducer is defined semantically in terms of an algebraic ***monoid***: a triple $(T, \otimes, e)$, where T is a set and $\otimes$ is an associative binary operation over T with identity e. From an object-oriented programming perspective, the set T forms the base type of a reducer's views, and the reducer provides a member function REDUCE that implements the binary operator $\otimes$ and a member function CREATE-IDENTITY that constructs an identity element of type T. The reducer also provides one or more UPDATE functions, which modify an object of type T. From the programmer's perspective, the reducer library provides a list of commonly used monoids; the programmer can also declare a reducer with a user-defined view type, so long as the view type implements an identity function (invoked by the CREATE-IDENTITY function) and a binary associative operator (invoked by the REDUCE function).

During parallel execution, the Cilk runtime supports parallel updates to a reducer by generating and maintaining multiple views for that reducer, thereby allowing each parallel subcomputation to operate on its own local view. In particular, when a worker first executes a UPDATE call to a reducer after a successful steal, it automatically calls CREATE-IDENTITY to create a new identity view of the reducer. Because a worker executes a Cilk computation in its serial order between successful steals, the worker can safely apply the call to UPDATE, as well as all subsequent calls to UPDATE, to this view, until it steals again. As stolen subcomputations return, the runtime automatically combines their corresponding views using the REDUCE operation, in the same order as how these updates would be applied in a serial execution. In the absence of a race, as long as the REDUCE operation is semantically associative, the resulting view is the same as if the program were run serially.

```
1  void update_list(int n, MyList<int>& list) {
2    cilk::reducer< list_monoid<int> >
         list_reducer;
3    list_reducer.set_value(list);
4    int x = cilk_spawn foo(n, list_reducer);
5    cilk_for(int i = 0; i < n; ++i) {
6      list_reducer.view().insert(i);
7    }
8    cilk_sync;
9    list = list_reducer.get_value();
10 }

12 void race(int n, MyList<int>& list) {
13   int length = 0;
14   MyList<int> copy(list);
15   length = cilk_spawn scan_list(list);
16   update_list(n, copy);
17   cilk_sync;
18   return;
19 }
```

Figure 1: Example Cilk program that contains a determinacy race on the reduce operation of a linked-list reducer.

How a view-read race can occur. To illustrate how a view-read race can occur, let us first consider the code for the `update_list` routine shown in Figure 1. The function `update_list` takes in as parameters an integer `n` and a user-defined `list` of type `MyList` that implements a singly linked list with a head and a tail pointer to enable fast list concatenation. The `update_list` routine spawns `foo` with `n` and `list_reducer` to perform some computation, which may execute in parallel with the continuation on lines 5–7, a parallel loop that inserts `n` elements into the linked list. To coordinate parallel accesses to the list, `update_list` wraps the given linked list in a reducer on line 2. Since the reducer has a user-defined view type, the programmer must also supply the functions for implementing the reducer's CREATE-IDENTITY and REDUCE operations, which are defined via the `list_monoid` type (actual implementation not shown in the pseudocode).

Assuming that `list_monoid` implements these functions correctly, `update_list` as written does not contain a determinacy race, since the runtime coordinates parallel updates to the linked list via the use of a reducer. The routine does not contain a view-read race, either, since the value of `reducer_list` is initialized at line 3 before anything is spawned, and the value of `reducer_list` is retrieved at line 9 after all spawned subcomputations that may use the reducer have returned. The code would have been a view-read race, however, if `get_value` is invoked, say, before `cilk_sync`, since at that point `foo` might be accessing the reducer in parallel.

How a determinacy race involving a reducer can occur. The code in Figure 1 can exhibit a determinacy race between the `race` and `update_list` routines, however. In this code, the `race` routine invokes `scan_list`, which iterates through the elements of `list` until one is found with a NULL pointer to the next element. This spawned `scan_list` invocation can run in parallel with its continuation, the call to `update_list` on line 16. Since `update_list` might actually insert into the list, the `race` routine makes a copy of the list first at line 14 and passes the copy to `update_list`, so as to allow the `scan_list` to scan the snapshot of the list without the new inserts performed by `update_list`. Unfortunately, this code contains a bug because the copy constructor on line 14 only performs a shallow copy. That means, even though `copy` is a new `MyList` object, created with its own distinct head and tail pointers, the two `list` and `copy` lists still point to the same set of linked-list elements, leading to a determinacy race in the code. In particular,

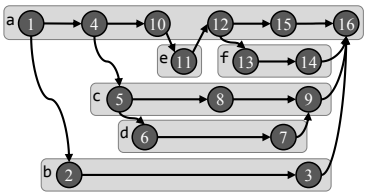

Figure 2: Example Cilk computation dag. Dark rectangles represent strands, edges represent parallel control dependencies between strands. The strands are labeled in their serial execution order. Light rectangles are labeled with the function instantiation and encompass the strands that execute within that instantiation.

whenever `scan_list` reaches the last linked-list node reading the `null next` pointer, some parallel subcomputation in `update_list` might be writing to that same `next` pointer to insert an element. This determinacy race means that the `scan_list` may scan a nondeterministic number of elements in the list.

Furthermore, where the determinacy race occurs is subtle. Because `update_list` employs a reducer to coordinate parallel inserts, any insert into the list can occur on a distinct view local to the subcomputation performing the insertion. What eventually writes to the `next` pointer and constitutes the race occurs is a REDUCE operation that eventually appends to the original view of `list_reducer`, as initialized in line 3. A tool such as Cilk Screen will not catch this particular race, because the determinacy race involves a view-aware instruction executed in a REDUCE operation.

3. THE PEER-SET ALGORITHM

This section presents the Peer-Set algorithm for detecting view-read races. View-read races are defined formally in terms of the "peer-set semantics" that reducers obey. We review the dag model of dynamic multithreading. We describe the "peer-set semantics" in terms of this dag computation model, and we formally define the view-read races based on these semantics. We describe the Peer-Set algorithm and how it checks for the "peer-set semantics."

The dag model for dynamic multithreading. We adopt the dag model for dynamic multithreading similar to the one introduced by Blumofe and Leiserson [5], which models a Cilk computation — the execution of a Cilk program — as a ***dag (directed acyclic graph)*** $A = (V, E)$, whose vertices are ***strands*** — sequences of one or more instructions containing no parallel control — and whose edges denote parallel control dependencies between strands. Figure 2 illustrates a Cilk computation dag that we shall use as a running example. The strands in Figure 2 are numbered in their ***serial execution order*** — the depth-first traversal of the dag in which every spawned child is visited before its continuation. We shall assume that strands respect boundaries of ***Cilk functions*** — functions that can spawn. That means, calling or spawning a Cilk function terminates a strand, as does returning from a Cilk function. Each strand thus belongs to exactly one Cilk function invocation. For now we shall not worry about modeling the executing of view-aware strands in the computation dag.

For any two strands u and v, we say that u ***precedes*** v, denoted as $u \prec v$, if there exists a path from u to v in the dag. Two strands u and v are logically in ***series*** if either $u \prec v$ or $v \prec u$; otherwise they are logically ***parallel***, denoted as $u \parallel v$. In Figure 2, for example, strands 4 and 9 are logically in series, because strand 4 precedes strand 9, while strands 9 and 10 are logically in parallel.

Peer-set semantics. "Peer-set semantics" dictate which updates are guaranteed to be reflected in the view of a reducer h observed at strand u in terms of the ***peers*** of u — the set of strands in parallel with u, denoted by $peers(u) = \{w \in V : w \parallel u\}$. Conceptually, "peer-set semantics" dictate that the view visible to a strand v is guaranteed to reflect the updates since a previous strand u if u and v have the same peers. In Figure 2, for example, these semantics dictate that the view of a reducer at strand 9 is guaranteed to reflect the updates since strand 5, because strands 5 and 9 have the same peers. The view at strand 14, meanwhile, is not guaranteed to reflect the updates since strand 10, because strands 10 and 14 do not share the same peers — strands 12 and 13 are in the peer set of strand 14, but not that of strand 10. The following definition formally defines ***peer-set semantics***:

DEFINITION 1 (PEER-SET SEMANTICS). *Let h be a reducer with an associative operator $\otimes$. Consider a serial walk of G, and let $a_1, a_2, \ldots, a_k$ denote the updates to h after the start of instruction u and before the start of instruction v. Let $h(u)$ and $h(v)$ denote the views of h at strands u and v respectively. If $peers(u) = peers(v)$, then $h(v) = h(u) \otimes a_1 \otimes a_2 \otimes \ldots \otimes a_k$.*

View-read races. Formally, a ***view-read race*** occurs when two accesses to reducers, called ***reducer-reads***, occur at strands with different sets of peers. Here, we broadly define a ***reducer-read*** as creating a reducer, resetting a reducer's value, or querying the reducer to retrieve its value. On the other hand, invoking CREATE-IDENTITY, UPDATE, or REDUCE on a reducer does not count as a reducer-read, because those functions operate on a reducer's underlying view instead of on the reducer itself. For example, consider the computation dag in Figure 2 and suppose that strands 1 and 9 read the value of the reducer. Because strands 1 and 9 do not share the same peer set, a view-read race exists between strands 1 and 9.

Given this definition of a view-read race, a Cilk program with a view-read race might nevertheless behave deterministically. For instance, in the code example shown in Figure 1, suppose that the programmer moves the call to `list_reducer.set_value(list)` to after `cilk_spawn` at line 4, thereby creating a view-read race. If `foo` does not modify `list`, however, then the `update_list` routine could behave deterministically, rendering the view-read race ***benign***. We nevertheless declare this to be a race because the reducer-reads violate their peer-set semantics.

The Peer-Set algorithm. The Peer-Set algorithm executes a Cilk computation serially and evaluates its strands in their serial execution order to check for view-read races. The Peer-Set algorithm employs several data structures to track which strands read the reducer and which strands have the same peer set.

During the execution, the Peer-Set algorithm assigns a unique ID to every Cilk function instantiation and maintains, for each instantiation F on the call stack, two scalars, $F.ls$ and $F.as$, and three bags, $F.SS$, $F.SP$, and $F.P$. Each bag stores a set of ID's for completed instantiations in a fast disjoint-set data structure [10, Ch. 21].

- The $F.as$ scalar stores the ***ancestor-spawn count*** — the total number of spawns that each ancestor F' of F has performed since F' last synced.
- The $F.ls$ scalar stores the ***local-spawn count*** — the number of spawns F has executed since F last synced.
- The $F.SS$ bag contains the ID's of all completed descendants of F with the same peer set as the first strand of F.
- The $F.SP$ bag contains the ID's of all completed descendants of F with the same peer set as the last continuation strand executed in F. If F has not spawned since it last executed a sync, then $F.SP$ is empty.
- The $F.P$ bag contains the ID's of all completed descendants of F not in $F.SS$ or $F.SP$.

For each Cilk frame F, we refer to the sum of the ancestor-spawn and local-spawn counts, $F.as+F.ls$, as the ***spawn count*** of F, which

F calls or spawns G:	G returns to F:
1 **if** F spawns G 2 $F.ls$ += 1 3 $F.P \cup= F.SP$ 4 $F.SP = \emptyset$ 5 $G.as = F.as + F.ls$ 6 $G.ls = 0$ 7 $G.SS =$ MakeBag(G) 8 $G.SP =$ MakeBag($\emptyset$) 9 $G.P =$ MakeBag($\emptyset$)	1 $F.P \cup= G.P$ 2 **if** F spawned G 3 $F.P \cup= G.SS$ 4 **elseif** $F.ls = 0$ 5 $F.SS \cup= G.SS$ 6 **else** 7 $F.SP \cup= G.SS$
F syncs:	F reads reducer h:
1 $F.ls = 0$ 2 $F.P \cup= F.SP$ 3 $F.SP =$ MakeBag($\emptyset$)	1 **if** FindBag($reader(h)$) is a P bag **or** $reader(h).s \neq F.as + F.ls$ 2 a view-read race exists 3 $reader(h) = F$ 4 $reader(h).s = F.as + F.ls$

Figure 3: Pseudocode for the Peer-Set algorithm. The MakeBag routine creates a new bag with a specified initial contents and a view ID. When passed $\emptyset$, MakeBag produces an empty bag. The FindBag routine finds the bag containing the specified element by finding the corresponding set in the disjoint-set data structure.

corresponds to the number of spawn statements executed by F and F's ancestors that have not been synced yet.

The Peer-Set algorithm also maintains a ***shadow space*** of shared memory, called *reader*, which maps each reducer to its last reader and the access context. That is, for each reducer h, $reader(h)$ stores the ID of the Cilk function F that last read h, and the associated field $reader(h).s$ stores the spawn count of F when it last read h.

Figure 3 gives the pseudocode of the Peer-Set algorithm, which maintains the bags and scalars for each function frame F as follows. When created, frame F inherits its ancestor-spawn count from the spawn count of its parent, and it initializes its local-spawn count $F.ls$ to 0. As F executes, it increments $F.ls$ when F spawns, and resets $F.l$ to 0 when F syncs. Frame F's bags are updated when a child frame G returns to F, based on whether F has spawned since it last synced. Although the bag $G.P$ is always combined with $F.P$, the bag $G.SS$ is combined with $F.SS$ only if F has not spawned since it last synced; otherwise $G.SS$ is combined with $F.SP$. The bag $G.SP$ is guaranteed to be empty when G returns to F because functions implicitly sync before they return in Cilk.

Given this algorithm, we can see that the Peer-Set algorithm has runs in the following time.

Theorem 1. *Consider a Cilk program that executes in time T on one processor and references x reducer variables. The Peer-Set algorithm checks this program execution for a view-read race in $O(T\alpha(x, x))$ time, where α is Tarjan's functiona, l inverse of Ackermann's function.*

Proof. Given the size of the shadow memory for the Peer-Set algorithm, the theorem follows from the analyses in [15]. □

4. CORRECTNESS OF THE PEER-SET ALGORITHM

This section provides discusses why the Peer-Set algorithm is correct. We provide intuition for how the Peer-Set algorithm properly detects view-read races. We argue mathematically for its correctness.

Intuition for the Peer-Set algorithm. To understand how the Peer-Set algorithm works, let us first consider the contents of the bags of a function F when F returns, considering the execution of the Peer-Set algorithm on the dag in Figure 2 as a running example. The bag $F.SP$ is empty, because in Cilk, a function F always syncs before it returns. Consequently, the bag $F.SS$ identifies descendants of F with the same peer set as the first instruction in F, and the bag $F.P$ contains all other descendants of F. In the example dag in Figure 2, then when c returns, bag c.SS contains the ID for c, and bag c.P contains the ID for d.

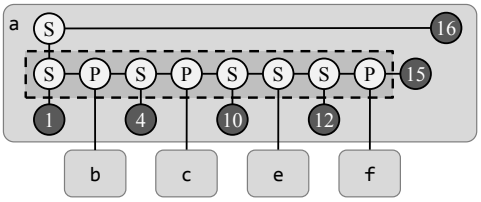

Figure 4: The canonical SP parse tree for the function instantiation a in the computation dag in Figure 2. The internal nodes of a sync block are indicated by the darkened rectangle outlined by a dashed line.

What happens to these bags when a function G returns to its parent F? The functions identified in $G.P$ must have a different peer set from that of any strand in F, and therefore, $G.P$ is always unioned with $F.P$. In the example dag in Figure 2, when c returns to a, unioning bag c.P with bag a.P correctly identifies that d has a distinct peer set from every strand in a.

As for $G.SS$, we must consider a few cases. Suppose that F spawned G. By definition of a spawn, all descendants of G must therefore have a different peer set from any strand in F. The bag $G.SS$ is thus unioned with the bag $F.P$ when G returns. In the example dag in Figure 2, because a spawned c at strand 4, every strand in c is in parallel with strand 10, implying that c has a distinct peer set from all strands in a.

If F called G when F had no outstanding spawned children, then the first strand in G has the same peer set as the first strand in F, and the bag $G.SS$ is therefore unioned with $F.SS$. Otherwise, F called G when F's local-spawn count was nonzero, meaning that F had at least one outstanding spawned child. The first strand in G therefore has a distinct peer set from that of the first strand in F, but the same peer set as the last continuation strand F executed. The $G.SS$ bag is therefore unioned with $F.SP$, where it remains until F either spawns again or syncs. In the example dag in Figure 2, strand 11 has a distinct peer set from strand 1, but the same peer set as strand 10, the caller of e. When e returns to a, therefore, unioning the bag e.SS with a.SP correctly identifies that the peer set of strand 11 matches that of strand 10.

Now let us consider detecting a view-read race. If a strand reads a reducer h, and $reader(h)$ is in some ancestor's P bag $F.P$, then it certainly has a different peer set. If $reader(h)$ is in $F.SS$ or $F.SP$ for some ancestor F, however, then $reader(h)$ has the same peer set as a strand u that is either F's first strand or the last continuation strand that F executed, and the currently executing strand v might have a distinct peer set from u. To handle this case, the Peer-Set algorithm compares the spawn count of v against the spawn count of $reader(h)$, stored in $reader(h).s$, which must match the spawn count as u. As long as v has this same spawn count, then no ancestor of v below F added a peer to v that is not a peer of u, meaning that u and v have the same peer set.

Correctness of the Peer-Set algorithm. To show why the Peer-Set algorithm works correctly, we model the computation dag using an "SP parse tree" as introduced by Feng and Leiserson [15]. As Feng and Leiserson show, the dag modeling a Cilk computation (that does not use reducers) is a ***series-parallel dag***, which has a distinguished ***source*** vertex s and a distinguished ***sink*** vertex t and can be constructed recursively with series and parallel compositions. This recursive construction can be represented by a binary tree, called an ***SP parse tree***.

Figure 4 illustrates the SP parse tree corresponding to function a in the dag in Figure 2. The leaves of the SP parse tree are strands in the dag, and each internal node is either an S node or a P node,

denoting either a series or parallel composition, respectively, of its two children. The SP parse tree in Figure 4 is a ***canonical*** parse tree [15], meaning that its internal nodes are laid out as follows. The sync strands in a Cilk function F partition the strands in F into ***sync blocks***. The canonical SP parse subtree for a sync block is a chain of S and P nodes, where the left child of each node is either a strand in F or the root of the canonical parse tree for a subcomputation spawned or called in F, and the right child is the next S or P node at the root of the SP parse subtree for the vertices following the left subchild in the serial order. A chain of S nodes, called the ***spine***, links the sync blocks exist within F.

To show that the Peer-Set algorithm is correct, we first show that two strands have the same peer set if and only if they are connected by S nodes in the SP parse tree.

LEMMA 2. *Two strands u and v have the same peer set, $peers(u) = peers(v)$, if and only if the path connecting u to v in the SP parse tree consists entirely of S nodes.*

PROOF. Let $\text{LCA}(u, v)$ denote the least-common ancestor of u and v in the SP parse tree. We first show that $\text{LCA}(u, v)$ must be an S node. If $\text{LCA}(u, v)$ is a P node, then $u \parallel v$, and therefore $u \in peers(v)$. Because $u \notin peers(u)$, we have that $peers(u) \neq peers(v)$.

Suppose that the path in the SP parse tree from $\text{LCA}(u, v)$ to u contains a P node. Then there must exist a strand w such that $\text{LCA}(u, w)$ is this P node, which implies that $u \parallel w$ and, therefore, that $w \in peers(u)$. Because this P node is on the path from $\text{LCA}(u, v)$ to u, we have that $\text{LCA}(w, v) = \text{LCA}(u, v)$, which is an S node. Therefore, $w \not\parallel v$, and thus $w \notin peers(v)$. This P node therefore implies that $peers(u) \neq peers(v)$, so if $peers(u) = peers(v)$, then no such P node can exist. A symmetric argument shows that no P node can exist on the path from $\text{LCA}(u, v)$ to v.

Now suppose that $peers(u) \neq peers(v)$. Without loss of generality, suppose that u executes before v in the serial order. If $v \in peers(u)$, then $u \parallel v$ and $\text{LCA}(u, v)$ is a P node. Otherwise, we have $u \prec v$ and there exists some strand w in exactly one of $peers(u)$ or $peers(v)$. Suppose that $w \in peers(u)$ and $w \notin peers(v)$. Then $w \parallel u$, implying that $\text{LCA}(w, u)$ is a P node [15, Lemma 4], and $w \not\parallel v$, implying that $\text{LCA}(w, v)$ is an S node. The nodes $\text{LCA}(w, u)$ and $\text{LCA}(w, v)$ therefore differ, and one can show that either $\text{LCA}(u, v)$ is one of these two. Either way, the P node $\text{LCA}(w, u)$ appears on the path from u to v in the SP parse tree. The case where $w \notin peers(u)$ and $w \in peers(v)$ is similar. □

Next, we argue that the Peer-Set algorithm identifies pairs of strands that are connected via S nodes in the SP parse tree. As in [15], we define the ***procedurification*** function F as the map from strands and nodes in the SP parse tree to Cilk function invocations.

LEMMA 3. *Consider an execution of the Peer-Set algorithm on a Cilk computation. Suppose that strand u executes before strand v, and let F be the procedurification function mapping the SP parse tree to Cilk function invocations. Let $a = \text{LCA}(u, v)$ be the least common ancestor of u and v in the SP parse tree. Then the both of the following conditions hold if and only if the path from u to v in the SP parse tree consists entirely of S nodes.*

- *The ID for $F(u)$ belongs to either the SS bag or the SP bag of $F(a)$ when v executes.*
- *The spawn count for $F(a)$ when u executes equals the spawn count for $F(v)$ when v executes.*

PROOF SKETCH. Bags $F(a).SS$ and $F(a).SP$ contain the set of descendants reachable from $F(a)$'s first strand and $F(a)$'s last-executed continuation strand, respectively, via S nodes in the parse tree. Either $F(a).SS$ or $F(a).SP$ contains u if and only if there is no P node along the path from u to a. Spawn counts allow us to check whether there is any P node along the path from a to v, because the spawn count of $F(u)$ when u executes matches that of $F(v)$ when v executes if and only if the path from a to v contains no P node. □

With Lemmas 2 and 3, we show that the Peer-Set algorithm detects a view-read race if only if one exists.

THEOREM 4. *The Peer-Set algorithm detects a view-read race in a Cilk computation if and only if a view-read race exists.*

PROOF. Let F be the procedurification function mapping the SP parse tree to Cilk function invocations.

We first see that, if the Peer-Set algorithm detects a view-read race, then one exists. If the Peer-Set algorithm detects a view-read race on a reducer h when executing strand e_2, then Figure 3 shows that either $reader(h)$ belongs to a P bag or the spawn count $reader(h).s$ does not match $F(e_2).as + F(e_2).ls$. Lemma 3 therefore implies that a P node exists on the path from e_1 to e_2 in the SP parse tree, meaning that $peers(e_1) \neq peers(e_2)$ by Lemma 2. Consequently, a view-read race exists.

We now see that, if a view-read race exists on a reducer h, then the Peer-Set algorithm detects it. Let e_1 and e_2 be two strands involved in a view-read race on reducer h, where e_1 executes before e_2 in the serial order and, if several such races exist, we choose the race for which e_2 executes earliest in the serial order. The definition of a view-read race implies that $peers(e_1) \neq peers(e_2)$.

When e_2 executes, suppose that $reader(h) = F(e)$ for some strand e. If $e = e_1$, then because $peers(e_1) \neq peers(e_2)$, Lemmas 2 and 3 imply that a view-read race is reported. If $e \neq e_1$, then e must have executed after e_1, in order to overwrite $reader(h)$. We must also have that $peers(e) = peers(e_1)$; otherwise Lemmas 2 and 3 imply that a view-read race exists between e and e_1, and that fact that both e and e_1 execute before e_2 contradicts strand e_2 being the earliest strand in the serial order for which a view-read race exists on h. Because $peers(e_1) \neq peers(e_2)$, we have that $peers(e) \neq peers(e_2)$, and by Lemmas 2 and 3, a view-read race is detected. □

5. THE SP+ ALGORITHM

This section presents the SP+ algorithm for detecting determinacy races in a Cilk computation that uses a reducer. For a Cilk program that uses reducers, its parallel execution contains view-aware strands and runtime-invoked REDUCE operations that generate additional ***reduce strands*** and corresponding dependencies. The SP+ algorithm extends the SP-bags algorithm [15] to handle these additional complexity in execution due to the use of reducers. We describe more precisely how the Cilk runtime manages views and how one can model these additional strands and dependencies using a "performance dag." We identify the circumstances in which a determinacy race can exist in a computation that uses reducers. We describe how the SP+ algorithm detects such determinacy races.

The SP+ algorithm makes the following assumptions. First, the SP+ algorithm takes a steal specification as input, which dictates which continuations to steal, when to create new views, and how to reduce views. The steal specification removes all nondeterminism in how the Cilk runtime manages the reducers, allowing the SP+ algorithm to consider a single execution of the Cilk program. Second, it assumes that the REDUCE, CREATE-IDENTITY, and UPDATE functions execute only serial code, which is typically the case in real programs, and thus the execution of one of these functions can be modeled with a single strand. Following the same terminology as in types of instructions, we refer to a strand that arises from executing one of these functions as a ***view-aware*** strand, and other strands in the computation are ***view-oblivious*** strands.

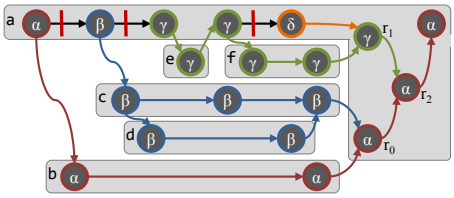

Figure 5: An example of performance dag, which corresponds to augmenting the user dag in Figure 2 in Section 2 with reduce strands r_0, r_1, and r_2. A vertical bar across an edge indicates that the following continuation strand is stolen. Each strand is labeled with its associated view ID. Strands with the same view ID are highlighted with the same color.

How the Cilk runtime manages views. To see how view-aware strands complicate determinacy race detection, let us first review how the Cilk runtime manages views in more detail. Let $h(u)$ denote the view of a reducer h seen by strand u in a dag $A = (V, E)$. The runtime system maintains the following ***view invariants***:

1. If u has out-degree 1 and $(u, v) \in E$, then $h(v) = h(u)$.
2. Suppose that u is a spawn strand with outgoing edges $(u, v), (u, w) \in E$, where u spawns v and leads to continuation strand w. Then, we have $h(u) = h(v)$ and either $h(w) = h(u)$ if w was not stolen, or $h(w)$ is a new view otherwise.
3. If $u \in V$ is a sync strand, then $h(u) = h(s)$, where s is the first strand of the Cilk function containing u.

When a new view $h(w)$ is created according to Invariant 2, we the new view $h(w)$ is a ***parallel view*** to $h(u)$. We say that the old view $h(u)$ ***dominates*** $h(w)$, which we denote by $h(u) > h(w)$. For a set H of views, we say that two views $h_1, h_2 \in H$ are ***adjacent*** if there does not exist $h_3 \in H$ such that $h_1 > h_3 > h_2$. Each parallel view created according to Invariant 2 is eventually destroyed by a call to Reduce. In particular, the runtime system always reduces adjacent pairs of views together, destroying the dominated view in the pair.

To maintain Invariant 3, the runtime ensures that all parallel views created within a sync block are reduced before the sync strand executes. One can show inductively that the views of the first and last strands of a function must be identical. This property holds because the first spawned child always inherits the view of the first strand, and a Reduce operation always reduces two adjacent views and destroys the dominated view. Moreover, the runtime always performs an implicit sync before a Cilk function returns.

The performance dag. A Cilk computation that uses a reducer can be modeled as a ***performance dag*** [27], which augments the ordinary computation dag with additional reduce strands from executing Reduce operations. The performance dag also modifies the dependencies going into a sync strand in order to incorporate the reduce strands and model the necessary dependencies among them. These dependencies among the reduce strands form a ***reduce tree*** before each sync node, with an additional dependency going from the root of the reduce tree into the corresponding sync node.

Figure 5 shows an example of a performance dag, which corresponds to augmenting the computation dag shown in Figure 2 with reduce strands. In this dag, three different continuation points are stolen, each causing a new view to be generated, leading to a total of 4 views in a. For each newly created view, there is a corresponding reduce strand produced from executing the Reduce operation that destroys the view, reducing its value into an adjacent view that dominates it. The reduce strand r_0, for example, reduces the views α and β, destroying β and inheriting the view ID α. Figure 5 also shows how these reduce strands form a reduce tree before the final strand, which is the sync strand in a. The reduce strands and the structure of the reduce trees are both functions of the execution schedule, which is fixed by the input steal specification.

Detecting determinacy races involving view-aware strands. View-aware strands complicate the circumstances under which a determinacy race occurs. For example, in the performance dag shown in Figure 5, let e_1 be the first strand in function d, and let e_2 be the second strand in function c. Suppose that e_1 and e_2 access the same memory location ℓ with one being a write, and suppose that e_2 is a view-aware strand generated from executing an Update. Because e_2 is a continuation that is not stolen in this execution, the same worker executes e_2 immediately after returning from d, and both e_1 and e_2 observe the same view β of the reducer, as Figure 5 shows. Because e_2 is view-aware, in a different execution in which e_2 is stolen, e_2 will observe a different view and might therefore write to a different memory locations. If location ℓ is part of view β, for example, then in this alternative scenario, e_2 might not write to ℓ, precluding a determinacy race with e_1. Because e_2 is view-aware, its logical parallelism with e_1 is not sufficient for it to definitively race with e_1; it must also operate on a parallel view.

We can summarize the conditions under which a determinacy race exists between two strands e_1 and e_2 in a Cilk computation that uses a reducer. Suppose that e_2 follows e_1 in the in serial execution order, both e_1 and e_2 access the same location ℓ, and at least one of them writes to ℓ.

- If e_2 is a view-oblivious strand, then a determinacy race exists between e_1 and e_2 if and only if e_1 and e_2 are logically in parallel.
- If e_2 is a view-aware strand, then a determinacy race exists between e_1 and e_2 if and only if e_1 and e_2 are logically in parallel and are associated with parallel views of a reducer.

The SP+ algorithm. Like the Peer-Set and SP-bags algorithms, the SP+ algorithm is a serial algorithm that evaluates the strands of a Cilk computation in their serial order. As it executes, SP+ employs several data structures to keep tracks of parallel views created according to the steal specification and determine the series-parallel relationships between strands, including reduce strands.

Like the SP-bags algorithm, SP+ maintains 2 shadow spaces of shared memory, called *reader* and *writer*. Each shadow space contains an entry for each memory location that the computation accesses. During the execution, each Cilk function instantiation is given a unique ID. Each location ℓ in *reader* stores the ID of the instantiation that last read ℓ, while each location ℓ in *writer* stores the ID for the instantiation that last wrote ℓ.

The SP+ algorithm also maintains a set of bags for each Cilk function F on the call stack. Each bag stores a set of ID's for completed procedures in a fast disjoint-set data structure. In particular, when executing a strand u, the bags associated with a function F on the call stack have the following contents:

- The ***S bag*** $F.S$ contains the ID's of F's completed descendants that precede u, as well as the ID for F itself.
- The ***P stack*** $F.P$ contains a stack of P bags. Together, the P bags in $F.P$ contain the set of ID's of F's completed descendants that are logically in parallel with u. The P bags $p \in F.P$ partition this set into subsets whose views are in series with the last ID to be added to the P bag p.

Figure 6 gives the pseudocode of the SP+ algorithm. Like the SP-bags algorithm, the SP+ algorithm pushes new S and P bags onto the call stack when is executes a function call or spawn, and it pops these bags off of the call stack when the function returns.

Unlike in the SP-bags algorithm, the SP+ algorithm also maintains P stacks. Conceptually, each P stack in SP+ replaces a P bag in the SP-bags algorithm in order to keep track of views. Each P bag p has an associated ***view ID***, denoted $p.vid$, which is a unique ID associated with the P bag on its creation. Executing a stolen continuation pushes a new P bag with a new view ID onto the top

To detect a potential race with a view-aware strand, SP+ checks that not only the two strands are in parallel, but that they also operate on parallel views, as verified by comparing the view IDs of the last access and currently executing strand. Note that the union of the top two P bags occurs *before* the invocation of the corresponding REDUCE operation, and thus any memory access performed by the reduce strand will have the same view IDs as the descendants in those P bags, achieving the desired effect — the reduce strand is in series with descendants in these two P bags.

There is one subtlety in how SP+ handles the shadow memory. SP+ replaces the last reader and writer only if the last access is in an S bag *or* if the current access is performed by a reduce strand and that it shares the same view as the last access. Call the last access in the shadow memory e_1, and the currently executing strand e_2. By "pseudotransitivity of $\|$" [15], we know that there is no need to replace e_1 in the shadow memory with e_2 if the e_2 is logically in parallel with e_1, because any strand that comes later in serial execution order that races with e_2 will race with e_1 as well. We need only to update the last reader / writer if e_2 is in series with e_1. In the case where e_2 is a reduce strand, however, e_2 is in series with e_1 even if e_1 belongs to a P bag but has the same view ID.

To illustrate how SP+ operates, consider it unfolding the performance dag shown in Figure 5 in Section 5. When it executes the fifth strand e in function a, the stolen continuation labeled with δ, it pushes a new empty P bag corresponding to view δ. The P stack a.P contains two other P bags: {b, c, d}, associated with view α, and {e, f}, associated with view γ. The first P bag resulted from unioning the P bags corresponding to views α and β before executing r_0. After SP+ executes e and encounters r_1, the steal specification dictates that the top two P bags — the empty one representing strand δ and the one containing {e, f} — are unioned before executing r_1. If r_1, a view-aware strand, happens to write to location ℓ last accessed by the first strand in f labeled with γ, SP+ will not report a race, since they now share the same view after the union. If the last access of ℓ before r_1 is performed by a strand in c, however, a race will be reported, since c is in a different P bag of a.

7. ANALYSIS OF THE SP+ ALGORITHM

This section discusses how the SP+ algorithm can be used to check if any execution on a given input of an ostensibly deterministic Cilk program that uses reducers contains a determinacy race involving a view-oblivious strand. If D is the Cilk depth and K is the maximum sync-block size (defined in Section 4), then we show that $\Omega(\max\{KD, K^3\})$ steal specifications are needed to elicit every possible view-aware strand, and $O(KD + K^3)$ steal specifications suffice. The proofs in this section can be adapted to construct these $O(KD + K^3)$ steal specifications.

The following theorem bounds the number of steal specifications needed to elicit all possible update strands to M, the maximum number of continuations in any sync block in a Cilk function. In a Cilk computation, if D is the Cilk depth and K is the maximum number of continuations in any sync block, then M can be as large as KD. The following theorem considers the Cilk computation's ordinary dag, not its performance dag.

THEOREM 6. *In a Cilk computation, all possible update strands can be elicited in $\Theta(M)$ steal specifications, where M is the maximum number of continuations not followed by a sync strand in the same Cilk function along any path in the computation dag.*

PROOF. Consider the canonical SP parse tree for the computation dag. Let a denote an internal node in this tree whose left child is l and whose right child is r. If a is an S node, then the subcomputation under r inherits the value of the view $h(l)$. Because the reducer is a monoid, the value of $h(l)$ is the same, regardless of how the subcomputation under l was scheduled. The same situation holds if a is a P node unless the subcomputation under r is stolen, in which case the subcomputation under r executes on a new, identity view. In this case, because the reducer is a monoid, the value of $h(e)$ does not depend on the computation executed before e.

Consider the root-to-e path p in the SP parse tree. From the argument above, the value of $h(e)$ depends only on the closest P node $a \in p$ such that the right child of a inherits an identity view. The number of different values of $h(e)$ is therefore the number of P nodes a in p for which e is in the right subtree of a.

A root-to-e path in the canonical SP parse tree passes through at most one sync block in each Cilk function F, and each P node in F on that path corresponds to a continuation on the path to e in that sync block. Consequently, $\Omega(M)$ steal specifications are needed to elicit all possible update strands at the location of e. Because there exists a unique path in the SP parse tree from the root to each strand e, continuations to steal can be chosen in a breadth-first manner, where two continuations e_1 and e_2 are stolen in the same specification if the same number of P nodes occur on the root-to-e_1 and root-to-e_2 paths in the tree. Consequently, $O(M)$ steal specifications suffice to elicit all possible update strands. □

We now consider the number of steal and reduce specifications needed to elicit all possible reduce strands. Every REDUCE operation on a sequence $\kappa = \langle k_1, k_2, \ldots, k_K \rangle$ of K elements combines two adjacent subsequences of κ. There are therefore $\binom{K}{3}$ distinct REDUCE operations on K, and therefore $O(K^3)$ specifications can elicit all possible reduce strands. The following theorem shows that, $\Omega(K^3)$ specifications are necessary to elicit every reduce strand.

THEOREM 7. *Let $\kappa = \langle k_1, k_2, \ldots, k_K \rangle$ be an ordered set of K elements. Any collection R of reduce trees on κ that contains each* REDUCE *operation at least once has size $|R| = \Omega(K^3)$.*

PROOF. To bound the number of reduce trees in R, let us characterize a REDUCE operation by the size of its larger input view. Each view h of a reducer corresponds to some subsequence of κ, and the ***size*** of h is the length of the subsequence corresponding to h. For example, a reduce strand that reduces the views represented by the subsequences $\langle k_a, k_{a+1}, \ldots, k_{b-1} \rangle$ and $\langle k_b, k_{b+1}, \ldots, k_{c-1} \rangle$ of κ reduces a view of size $b - a$ with one of size $c - b$. Let us consider reduce strands for which the size of its larger input is at least $n/2+1$.

To count the number of reduce trees containing such reduce strands, we imagine iteratively constructing the collection R of reduce trees by considering different view sizes in increasing order. For each size s, each view h of size s can be an input to multiple distinct possible reduce strands. Because $s \geq n/2 + 1$, each reduce tree in R can contain at most one view h of size s and at most one reduce strand r on such a view. A reduce tree in R that produces h might already contain r already; otherwise a new reduce tree must be added to R that contains r.

We can lower bound the number of reduce trees added to R for each size s using the following observations:

- There are $n - s + 1$ distinct views of size s.
- For each view h of size s, there are $n - s$ distinct reduce strands that take h as an input.
- For each view h of size s, at most 2 reduce trees in R can produce h from a smaller view of a particular size s', where $n/2 + 1 \leq s' < s$. Consequently, there are at most $2(s - n/2 - 1)$ reduce trees already in R that contain distinct reduce strands on h.

These observations show that, for a particular size $s \geq n/2 + 1$, there are $(n - s + 1)(n - s)$ different reduce strands on views of size s, and at most $(n - s + 1)2(s - n/2 - 1)$ of these reduce strands

can be exist in reduce trees already in R. For each size s, we must therefore add at least $(n - s + 1)(2n - 3s + 2)$ new reduce trees to R. This bound holds as long as $2n - 3s + 2 > 0$, implying that $s < 2(n + 1)/3$. Summing over the applicable sizes s, we have that $|R| \geq \sum_{s=n/2+1}^{2(n+1)/3-1}(n - s + 1)(2n - 3s + 2) = \Omega(n^3)$. □

8. RADER

This section presents Rader, our prototype implementation of Peer-Set and SP+. We evaluated Rader on 6 benchmarks. When running the Peer-Set algorithm, Rader incurs a geometric-mean multiplicative overhead of 2.32 (with a range of 1.03 – 5.95) over running the benchmarks without instrumentation. When running the SP+ algorithm, Rader incurs an overhead of 16.76 (with a range of 3.94 – 75.60). To get a sense of how much of the overhead comes from the instrumentation versus algorithm implementation, we measured the overhead of Rader over an empty tool (i.e., same instrumentation which leads to an empty call). When running the Peer-Set algorithm, Rader incurs a geometric-mean multiplicative overhead of 1.84 (with a range of 1 – 3.89) over running the benchmarks with an empty tool. When running the SP+ algorithm, Rader incurs an overhead of 7.27 (with a range of 3.04 – 15.68). This average is without considering `ferret`, an outlier that has very little overhead, which we explain later in the section.

Implementation. Rader uses compiler instrumentation to detect races in Cilk programs. We modified GCC 4.9 to insert instrumentation to identify parallel control constructs in the execution, akin to the Low Overhead Annotations [21] for Intel's Cilk Plus compiler.

For the SP+ algorithm, we also need to instrument reads and writes and simulate parallel executions. Rader instruments reads and writes by piggybacking on the ThreadSanitizer instrumentation [41], supported since GCC 4.8 [20]. When running SP+, Rader triggers operations in the runtime system to simulate steals at program points specified in a given steal specification. To accomplish this, Rader appropriately "promotes" various runtime data structures that would be modified if, after a worker executes the corresponding spawn, the parent of that spawn had been stolen [18]. When the worker resumes the parent later, it acts as if it stole the parent, and appropriately creates a new reducer view for the continuation. These promoted data structures also prompt the worker to check if it should execute any reduction.

Since Rader needs to check particular reductions according to the steal specification, the worker may need to hold off on a reduction instead of reducing eagerly, which is how Cilk runtime normally operates. We modified the runtime so that the worker, when simulating steals, calls back to Rader to see if it should execute a reduction. Although the modified runtime no longer always performs reduction eagerly, we optimized the steal specifications that Rader uses as follows to use only constant space per steal.

Steal specifications. Although constructing the steal specification naively can cause the input to be as large as the computation dag, one can do better. Because Section 7 showed that $\Omega(\max\{KD, K^3\})$ executions are necessary to guarantee completeness and that $O(DK + K^3)$ suffice, no time is saved asymptotically if the system checks for more than one particular reduction or update per sync block. We therefore only need to make sure that Rader checks at least one reduction or update per sync block in a given execution. Consequently, the steal specification can be as simple as specifying which three continuations to steal in a sync block (for checking reduce operations) or which continuations at a particular depth to steal (for checking updates). We can steal the same continuations for every sync block, and the completeness guarantee still stands, as long as we run Rader with $O(K^3)$ different specifications. In practice, Rader takes as an input either three values specifying the continuations to be stolen, or a random seed and the maximum sync block size, in which case three different points are chosen randomly for each sync block. If a race is detected, Rader reports the labels corresponding to the stolen continuations that triggered the race, making it easy to repeat the run for regression tests.

Experimental evaluation. We empirically evaluated Rader on 6 benchmark applications, which Figure 7 lists. We converted the pipeline programs `dedup` and `ferret` from the PARSEC benchmark suite [4] to use Cilk linguistics and a `reducer_ostream` (part of Cilk Plus) to write its output. The synthetic `fib` benchmark uses a `reducer_opadd`, which is also part of Cilk Plus. All other benchmarks use user-defined reducers, including a "Bag" data structure (`pbfs` [27]), a "hypervector" (`collision`), and a user-defined struct (`knapsack` [17]). All experiments ran serially on an Intel Xeon E5-2665 system with 2.4 GHz CPU's and 32 GB of main memory. Each core has a 32-KB private L1-data-cache and a 256 KB private L2-cache, and shares a 20 MB L3-cache with 7 other cores.

Figure 7 shows the overhead of Rader over running each benchmark without instrumentation. As Figure 7 shows, the Peer-Set algorithm (column *Check view-read race*) incurs little overhead. The SP+ algorithm has a somewhat high overhead for `fib` and `knapsack`, but otherwise exhibits reasonable overheads. The high overheads on these two benchmarks stems from the fact that they perform very little work per strand, making the overall running time dominated by instrumentation. We have also compared Rader's overhead over running each benchmark with an empty tool to gauge how much overhead comes from instrumentation versus algorithm implementation. As can be seen in Figure 8, the overhead due to algorithm implementation is minimal. The overhead dropped from as high as 75.60 down to 13.85 on `fib`, for example.

In terms of the additional bookkeeping for checking reducers (i.e., comparing columns `Check reductions` and `No steals`), most applications exhibit negligible overhead; `fib` exhibits the highest overhead in this regard (2.25 times overhead), because `fib` is a synthetic benchmark we devised to stress test Rader— each function call does almost no work except for updating reducers and reducing views. The overhead is thus evident — there is not much work to amortize it against. One interesting outlier is `ferret`, which has virtually no overhead. It turns out that, when we instrument all the library code that came with PARSEC, lots of races get reported. We separately confirmed with this Cilk Screen using Intel compiler, and indeed there were races. Since the reporting of races (I/O) throws off timing, we opt to instrument only the main `ferret` code without the rest of the library, meaning that only a small fraction of memory accesses within the computation are instrumented.

9. RELATED WORK

Race detection is a rich area actively being worked on. Roughly speaking, approaches to race detection can be categorized as either static [1, 2, 6, 13, 30, 36, 44] or dynamic [8, 9, 11, 14, 16, 33, 35, 40, 43, 46]. We focus our discussion on the dynamic approach, the category our work falls under. In particular, we shall focus on related work that supports a similar language model, namely work on detecting determinacy races in programs with nested parallelism.

Nudler and Rudolph [32] proposed a ***English-Hebrew labeling*** scheme that labels "parallel tasks" in a computation based on two different traversal orders, such that comparing the labels suffice to tell whether the two tasks are logically in parallel. This scheme uses static labels, meaning that, once assigned, the labels do not change, and the label size can grow proportionally to the maximum number of fork points in the program (i.e., execution points where parallel branches are spawned off).

			Overhead over no instrumentation			
Benchmark	*Input size*	*Description*	*Check view-read race*	*No steals*	*Check updates*	*Check reductions*
`collision`	20	Collision detection in 3*D*	1.03	17.25	17.11	17.10
`dedup`	*medium*	Compression program	1.21	6.72	6.71	6.67
`ferret`	*large*	Image similarity search	1.00	2.25	2.25	2.25
`fib`	28	Recursive Fibonacci	5.95	33.58	36.90	75.60
`knapsack`	26	Recursive knapsack	2.70	49.24	56.41	66.79
`pbfs`	$\|V\|$ = 0.3M, $\|E\|$ = 1.9M	Parallel breadth-first search	3.34	3.94	3.94	5.65

Figure 7: Rader's overhead over running 6 benchmarks without instrumentation. Both Rader and the benchmarks are compiled with `-O3`. We ran Rader with different configurations. *Check view-read race* shows the overhead when running the Peer-set algorithm for checking view-read races only. *No steals* shows the overhead of checking races without eliciting any reductions. *Check reductions* shows the overhead with randomly chosen steal points to eilcit subset of possible reductions. *Check updates* shows the overhead with steals at continuation depth that's half of the maximum sync block size.

			Overhead over an empty tool			
Benchmark	*Input size*	*Description*	*Check view-read race*	*No steals*	*Check updates*	*Check reductions*
`collision`	20	Collision detection in 3*D*	1.00	8.19	8.13	8.12
`dedup`	*medium*	Compression program	1.22	6.53	6.52	6.48
`ferret`	*large*	Image similarity search	1.00	1.04	1.04	1.04
`fib`	28	Recursive Fibonacci	3.89	6.15	6.76	13.85
`knapsack`	26	Recursive knapsack	2.44	11.56	13.24	15.68
`pbfs`	$\|V\|$ = 0.3M, $\|E\|$ = 1.9M	Parallel breadth-first search	1.79	3.04	3.04	4.6

Figure 8: Rader's overhead over running 6 benchmarks with an empty tool, i.e., instrumentation leads to empty calls. Both Rader and the benchmarks are compiled with `-O3`. We ran Rader with different configurations as described in Figure 7.

Dinning and Schonberg [12] proposed ***task-recycling scheme*** that improves upon the English-Hebrew labeling scheme by recycling labels for tasks, at the expense of failing to detect some races. They empirically demonstrated that the task-recycling scheme can be implemented efficiently.

Mellor-Crummey [29] proposed a different labeling scheme called ***offset-span labeling***, where the label sizes grow proportionally to the nesting depth, improving on the bound of the English-Hebrew labeling scheme. He also observed that, for parallel determinacy race detection, it suffices to keep only two readers in shared memory, namely, the "left-most" and "right-most" parallel readers, which are the least and most recent reads in the serial execution order of the computation.

Feng and Leiserson proposed the SP-bags algorithm [15], which employs a disjoint-set data structure to maintain series-parallel relationships. SP-bags executes the computation serially and incurs near-constant overhead per check. They also observed that, the parallel relationship is pseudotransitive, and thus it suffices to store only a single reader in the shadow memory.

Bender et. al proposed the SP-hybrid algorithm [3] that employs a scheme similar to English-Hebrew labeling, but manages the labels in a concurrent order-maintenance data structure, which allows dynamic labels and supports checks with constant overhead.

Raman et. al proposed ESP-bags [37] algorithm, which is similar to the SP-bags algorithm but extended to handle `async` and `finish` in Habanero-Java [7]. They subsequently proposed SPD3 detectors [38], also for Habanero-Java that maintains series-parallel relationships by keeping track of the entire computation tree, which has a simple implementation and executes in parallel.

Since our algorithms extend upon the SP-bags algorithm and similarly use a disjoint-set data structure, they enjoy similar time and space bounds; the SP+ has the additional overhead for simulating steals and reductions. Like SP-bags, however, they execute the computation serially. One distinct difference between our work and these algorithms is that SP+ handles race detection on computations with reducers, which correspond to non-series-parallel dags. As far as we know, the SP+ algorithm is the first determinacy race detector that provides provable guarantees for computations that are not series-parallel. Nevertheless, the SP+ algorithm, albeit sound for a given execution, requires polynomial number of executions to guarantee complete coverage, due to the inherent nondeterminism in how the runtime manages reducers.

10. CONCLUSION

We have presented two algorithms for catching two unique types of races that arise from incorrect use of reducer hyperobjects. Both algorithms are provably efficient and correct with respect to a given execution, and incur modest overhead in practice. We have also shown that for an ostensibly deterministic Cilk program, polynomially many SP+ executions with different steal specifications suffice to elicit all possible view-aware strands, thereby providing the desired coverage.

Both algorithms execute the computation serially, however, and a natural question is whether they can be parallelized to execute Cilk computations in parallel, so as to achieve better execution time for race detection. In particular, the Peer-Set algorithm has demonstrated negligible overhead when running serially; an efficient parallel algorithm can lead to a light-weight always-on view-read race detection tool. Here, we lay out some of the challenges that we foresee in parallelizing these algorithms.

To parallelize the Peer-Set algorithm, one challenge is figure out what minimal information needs to be stored in the shadow memory to correctly detect view-read races. The Peer-Set algorithm maintains the shadow memory to keep track of last readers in order to properly check whether two reads to a given reducer have the same peer set. If the algorithm executes the computation in parallel, there is no longer a clear notion of the last reader. For detecting determinacy races in parallel, Mellor-Crummey has demonstrated that, it is sufficient to store only two "left-most" and "right-most" parallel readers [29]. Such a scheme works for detecting accesses that are logically in parallel, but it is unfortunately insufficient for checking for peer-set equivalence. Storing all parallel reads encountered, meanwhile, incurs non-constant space usage per reducer and time overhead per check.

The main challenge to an efficient parallel SP+ algorithm, on the other hand, is to achieve the desired time bound so that one can get speedup during parallel execution. Recall that the SP+ algorithm executes the computation according to the steal specification, which dictates what continuations to steal and what reduce operations to execute in what order. The constraints imposed by the

steal specification mean that some worker threads may need to be blocked at certain execution points, which may have adversarial effects on load balancing. Conforming to the steal specification while maintaining good load balance seems to be an obstacle.

Acknowledgments

We thank Charles Leiseron of MIT for his help developing the peer-set semantics of reducers and for helpful discussions concerning determinacy races involving reducers. We thank William Leiserson of MIT for helpful discussions concerning types of determinacy races involving reducers. We thank Jeremy Fineman of Georgetown University and the Supertech group at MIT CSAIL for helpful discussions. We thank the reviewers for their excellent feedback.

11. REFERENCES

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *ACM TOPLAS*, 28(2):207–255, Mar. 2006.

[2] R. Agrawal and S. D. Stoller. Type inference for parameterized race-free java. In B. Steffen and G. Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *LNCS*, pp. 149–160. Springer Berlin Heidelberg, 2004.

[3] M. A. Bender, J. T. Fineman, S. Gilbert, and C. E. Leiserson. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *SPAA '04*, pp. 133–144, 2004.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT '08*, October 2008.

[5] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 1999.

[6] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *OOPSLA '01*, pp. 56–69. ACM, 2001.

[7] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-Java: the new adventures of old X10. In *PPPJ '11*, pp. 51–61, 2011.

[8] G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark. Detecting data races in Cilk programs that use locks. In *SPAA '98*, 1998.

[9] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI '02*, pp. 258–269. ACM, 2002.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[11] J. Devietti, B. P. Wood, K. Strauss, L. Ceze, D. Grossman, and S. Qadeer. RADISH: Always-on sound and complete race detection in software and hardware. In *ISCA '12*, pp. 201–212. IEEE Computer Society, 2012.

[12] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *PPoPP '90*, pp. 1–10, 1990.

[13] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP '03*, pp. 237–252. ACM, 2003.

[14] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *OSDI '10*, 2010.

[15] M. Feng and C. E. Leiserson. Efficient detection of determinacy races in Cilk programs. *TOCS*, 1999.

[16] C. Flanagan and S. N. Freund. Fasttrack: efficient and precise dynamic race detection. In *PLDI '09*, pp. 121–133, Dublin, Ireland, 2009. ACM.

[17] M. Frigo. A Cilk++ program for the knapsack challenge. `https://software.intel.com/en-us/courseware/249567`, 2009.

[18] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *SPAA '09*, pp. 79–90, 2009.

[19] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI '98*, 1998.

[20] GCC 4.8. GCC 4.8 release series changes, new features, and fixes. `https://gcc.gnu.org/gcc-4.8/changes.html`, 2014.

[21] Intel Corporation. Intrinsics for low overhead tool annotations. `https://www.cilkplus.org/open_specification/intrinsics-low-overhead-tool-annotations-v10`, Nov. 2011.

[22] Intel Corporation. *Intel® Cilk™ Plus Language Extension Specification, Version 1.1*. Intel Corporation, 2013. Document 324396-002US.

[23] Intel Corporation. An introduction to the Cilk Screen race detector. `https://software.intel.com/en-us/articles/an-introduction-to-the-cilk-screen-race-detector`, Apr. 2013.

[24] I.-T. A. Lee, S. Boyd-Wickizer, Z. Huang, and C. E. Leiserson. Using memory mapping to support cactus stacks in work-stealing runtime systems. In *PACT '10*, pp. 411–420. ACM, 2010.

[25] I.-T. A. Lee, A. Shafi, and C. E. Leiserson. Memory-mapping support for reducer hyperobjects. In *SPAA '12*, pp. 287–297, 2012.

[26] C. E. Leiserson. The Cilk++ concurrency platform. *Journal of Supercomputing*, 51(3):244–257, March 2010.

[27] C. E. Leiserson and T. B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *SPAA '10*, June 2010.

[28] D. McCrady. Avoiding contention using combinable objects. Microsoft Developer Network blog post, Sept. 2008.

[29] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of Supercomputing'91*, pp. 24–33, 1991.

[30] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *PLDI '06*, pp. 308–319. ACM, 2006.

[31] R. H. B. Netzer and B. P. Miller. What are race conditions? *ACM LOPLAS*, 1(1):74–88, March 1992.

[32] I. Nudler and L. Rudolph. Tools for the efficient development of efficient parallel programs. In *ICCSE*, 1986.

[33] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *PPoPP '03*, pp. 167–178. ACM, 2003.

[34] OpenMP Architecture Review Board. OpenMP application program interface, version 4.0. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`, 2013.

[35] E. Pozniansky and A. Schuster. MultiRace: Efficient on-the-fly data race detection in multithreaded c++ programs: Research articles. *Journal of CCPE*, 19(3):327–340, Mar. 2007.

[36] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: Practical static race detection for c. *ACM TOPLAS*, 33(1):3:1–3:55, Jan. 2011.

[37] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav. Efficient data race detection for async-finish parallelism. In *Runtime Verification*, volume 6418 of *LNCS*, pp. 368–383. 2010.

[38] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav. Scalable and precise dynamic datarace detection for structured parallelism. In *PLDI '12*, pp. 531–542, 2012.

[39] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Inc., 2007.

[40] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic race detector for multi-threaded programs. In *SOSP '97*, Oct. 1997.

[41] K. Serebryany and T. Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *WBIA '09*, pp. 62–71. ACM, 2009.

[42] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer. Phaser accumulators: a new reduction construct for dynamic parallelism. In *IPDPS '09*, 2009.

[43] C. von Praun and T. R. Gross. Object race detection. In *OOPSLA '01*, pp. 70–82. ACM, 2001.

[44] J. W. Voung, R. Jhala, and S. Lerner. Relay: Static race detection on millions of lines of code. In *ESEC-FSE '07*, pp. 205–214, Dubrovnik, Croatia, 2007. ACM.

[45] M. Wimmer. Wait-free hyperobjects for task-parallel programming systems. In *IPDPS '13*, pp. 803–812, 2013.

[46] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP '05*, pp. 221–234. ACM, 2005.