Efficient Race Detection for Reducer Hyperobjects

I-TING ANGELINA LEE, Washington University in St. Louis
TAO B. SCHARDL, MIT CSAIL

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 reducers.

We identify two unique types of races that arise from incorrect use of reducers in a Cilk program, and we present two algorithms to catch these races. 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 2.56 over running the benchmark without instrumentation. When running SP+, Rader incurs a geometric-mean multiplicative overhead of 16.94.

CCS Concepts: • **Software and its engineering → Software notations and tools**; **Concurrent programming structures**; **Software testing and debugging**; *Parallel programming languages*;

Additional Key Words and Phrases: Cilk; determinacy race; nondeterminism; reducers; view-read race

ACM Reference Format:
I-Ting Angelina Lee and Tao B. Schardl. 2017. Efficient Race Detection for Reducer Hyperobjects. *ACM Trans. Parallel Comput.* 9, 4, Article 39 (March 2017), 40 pages.
https://doi.org/0000001.0000001

1 INTRODUCTION

A fundamental challenge in multithreaded programming is to perform parallel updates to shared variables safely. A ***race*** is a common type of programming error, which occurs 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.

Many modern concurrency platforms provide some form of ***reduction mechanism*** [19, 23, 26, 29, 35, 40, 43, 47] 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

This work was supported in part by the National Science Foundation under Grant 1314547. Tao B. Schardl is supported in part by an MIT Akamai Fellowship and an NSF Graduate Research Fellowship.
Author's addresses: I-T. A. Lee (angelee@wustl.edu), 1 Brookings Drive, Campus Box 1045, St. Louis, MO 63130; T. B. Schardl (neboat@mit.edu), MIT CSAIL, 32 Vassar Street, Cambridge, MA 02139.

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 [19], provide a general reduction mechanism for Cilk programs. Reducers are supported by many Cilk dialects, including Intel Cilk Plus [23], Cilk++ [27], and Cilk-M [25]. Reducers exhibit several useful properties:

- Reducers are not tied to any particular linguistic construct and can be used in arbitrary Cilk code.
- 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.
- A reducer's update and reduce operations do not need to be commutative to produce a deterministic result; associativity suffices.

These properties are unique to reducers and make them a powerful general-purpose reduction mechanism. In contrast, OpenMP's reduction clause [35] is tied to its parallel loop constructs, and Microsoft's PPL's combinable objects [29] require reductions to be commutative.

Even though reducers are designed to coordinate parallel updates to shared variables, incorrect use of a reducer may nonetheless lead to races and cause a program to behave nondeterministically. A race involving the use of a reducer is challenging to debug because such a race exposes the nondeterminism in how the underlying runtime system manages views for reducers. Specifically, the Cilk runtime system employees two optimizations in order to manage views of reducers efficiently [19]. First, a new reducer view is created lazily, when actual parallelism is realized during execution. 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 can differ from run to run, depending on how the scheduling plays out.

As Section 3 illustrates with examples, incorrect uses of a reducer give rise to two unique types of races, which are difficult for the programmer to detect.

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 parallel subcomputations that might update the reducer have necessarily returned. Because the Cilk runtime system creates and reduces views based on scheduling, a view-read race can cause multiple runs of the same program to produce different results.

A second type of race, called a ***determinacy race*** [16] (also called a ***general race*** [32]), occurs when two logically parallel instructions operate on the same memory location, and at least one of them is a write. Provably efficient algorithms have been developed to detect determinacy races for Cilk [3, 16, 17, 44] and other concurrency platforms [12, 38, 39]. These race detectors provide correctness guarantees for finding races that involve ***view-oblivious*** instructions, which read and write memory locations that are not part of a reducer view. A Cilk program that uses a reducer, however, can also contain a determinacy race involving a ***view-aware*** instruction — an instruction that reads or writes a view of a reducer (e.g., an instruction that updates a view or reduces two views together). A race involving a view-aware instruction is particularly challenging to debug, because the view-aware instruction might not execute at all if the runtime system schedules the computation differently and manages the views differently as a result.

Extending existing race-detection algorithms to handle reducers while providing provable guarantees is non-trivial for two reasons. First, the use of a reducer generates parallel control dependencies that violate the structural assumptions that these algorithms depend on. Second, different runs of a Cilk program that uses a reducer can generate different schedules and thus different view-aware

instructions. Providing complete coverage could potentially require executing exponentially many different schedules to elicit all possible view-aware instructions.

Contributions

This paper addresses the question of how to efficiently and correctly detect races in Cilk programs that use reducers. Throughout the rest of the paper, when we talk about a Cilk program, implicitly we mean a Cilk program for a specific given input. When we talk about a Cilk computation, on the other hand, we mean an instance of an execution of a Cilk program with a given input and a specific schedule. This paper makes the following contributions.

The Peer-Set algorithm. We present the Peer-Set algorithm, which executes a Cilk program 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 program with the given input 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 a Cilk computation that uses reducers. Like the Peer-Set algorithm, SP+ executes the computation serially, although it simulates a particular parallel schedule in order to elicit a subset of view-aware instructions. The SP+ algorithm is provably correct for a given schedule, meaning it reports a determinacy race in a 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 one processor, M is the number of views created in the given schedule, and τ is the worst-case running time of a reduce operation. Compared to the SP-bags algorithm [16], an efficient sequential algorithm for detecting determinacy races in Cilk programs that do not employ reducers, the SP+ algorithm incurs additional overhead only to simulate a parallel schedule and execute reduce operations.

The Rader race-detection tool. We have developed a prototype tool, called Rader, that implements both the Peer-Set and SP+ algorithms to debug Cilk programs that use reducers. Rader implements these algorithms by using compiler-inserted program instrumentation to track memory accesses and parallel control dependencies. Using Rader, we demonstrate the efficiency of both algorithms in practice. We ran Rader on six application benchmarks that use reducers. Compared to running each benchmark sequentially without instrumentation, Rader incurred geometric-mean multiplicative overheads of 2.56 and 16.94 to run the Peer-Set and SP+ algorithms, respectively.

Analysis of the SP+ algorithm's coverage guarantees. 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 possible view-aware instructions. Although an exponential number of schedules exist for a given Cilk program, one can do better for *most* Cilk programs. Many 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 only employs reducers with semantically associative reduce operations. For such Cilk programs, we show how to construct a polynomial number of schedules to elicit all possible view-aware instructions. Given these schedules, one can then use the SP+ algorithm 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 on Cilk dynamic multithreading and reducers. Section 3 provides examples of a view-read race and a determinacy race that involves view-aware instructions. Sections 4 and 5 present the Peer-Set algorithm, along with intuition and a formal proof for its correctness. Section 6 presents the SP+

algorithm and a high-level intuition for its correctness. Section 7 introduces two concepts, the "spawn parse tree" and the "view parse tree," that are used in the formal proof of the SP+ algorithm's correctness. Section 8 formally shows that the SP+ algorithm correctly detects determinacy races. Section 9 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 10 describes our prototype implementation of Rader and empirically evaluates its performance. Section 11 discusses related work and Section 12 provides concluding remarks.

2 PRELIMINARIES

This section reviews the necessary background on Cilk and reducers. We describe Cilk linguistics and semantics, including that of reducers. We discuss how one models the execution of a Cilk program that does not use reducers. We summarize how a work-stealing scheduler schedules a Cilk computation and describe how the runtime supports parallel updates to reducers.

Cilk linguistics

Cilk extends C/C++ with the keywords `cilk_spawn`, `cilk_sync`, and `cilk_for`. 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 ***syncs***, 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 where all loop iterations may run in parallel with each other, and the control does not move beyond the end of the `cilk_for` until all iterations have completed. In practice, `cilk_for` is a syntactic sugar which compiles down to a call to a runtime function that uses `cilk_spawn` and `cilk_sync` to perform binary spawning of the iteration space and then to wait for iterations to complete. Every Cilk program has a serial counterpart, called its ***serial elision***, which can be obtained by removing all `cilk_spawn` and `cilk_sync` keywords, and replacing `cilk_for` with a regular **for** . In the absence of a race, a Cilk program is deterministic and produces the same result as its serial elision.

A Cilk program can coordinate parallel updates to a shared variable by declaring that variable to be a reducer. 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. A reducer may also provide one or more UPDATE functions, which modify an object of type T. From the programmer's perspective, the reducer library provides a set 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).

Figure 1 presents an example of a Cilk program that uses a reducer. In Figure 1a, we have a simple function `update_list` that uses the Cilk parallel constructs and a reducer that operates on `llist`, a user-defined view type. The class `llist` implements a singly-linked list with head and tail pointers to enable fast list concatenation (full implementation now shown). The function `update_list` first spawns function `foo` that performs some computation based on `n` and may insert

a)

```
01 void update_list(int n, llist<int>& ll) {
02   reducer< list_monoid > ll_reducer;
03
04   ll_reducer.set_value(ll);
05   int x = cilk_spawn foo(n, ll_reducer);
06   cilk_for (int i = 0; i < n; ++i) {
07     ll_reducer.view().insert(i);
08   }
09   cilk_sync;
10   ll = ll_reducer.get_value();
11 }
```

b)

```
12 class list_monoid
13     : public monoid_with_view< llist<int> > {
14 public:
15   typedef llist<int> value_type;
16   void identity(llist<int> *l) const {
17     new ((void*)l) llist<int>();
18   }
19   void reduce(llist<int> *l, llist<int> *r) const {
20     l->concat(r);
21   }
22 };
```

Fig. 1. Example Cilk program that uses a reducer with a user-defined view type. **a)** The Cilk program that uses a reducer with the user-defined view type `llist<int>`. This Cilk program declares a Cilk reducer with the type `list_monoid`. **b)** The definition of the `list_monoid` reducer type. This user-defined reducer type allows the reducer to operate on views of type `llist<int>`.

results into the list. Since `foo` is spawned, the continuation of `foo` — a parallel loop that inserts `n` elements into the list (lines 6–8) — may proceed in parallel with `foo`'s execution.

To coordinate parallel accesses to the list, `update_list` wraps the given linked list in a reducer on line 2. The `set_value` call on line 4 sets the initial value of the reducer to the input list `ll` before spawning `foo`, and the call on line 10 retrieves the final value after `cilk_sync`. Note that even though the end of `cilk_for` waits for all iterations to return, a `cilk_sync` on line 9 is necessary, because otherwise the execution may reach the `get_value` call before `foo` returns.

Figure 1b shows the `list_monoid` class that defines the reducer monoid with the `llist` view type. Because the reducer uses a user-defined view type, the programmer must specify the monoid for the reducer explicitly, by implementing the reducer's CREATE-IDENTITY and REDUCE operations. For `list_monoid`, lines 16–18 implement CREATE-IDENTITY, and lines 19–21 implement REDUCE.

The reducer coordinates parallel updates to the list and produces its final value that reflects what one would obtain by running the program's serial elision, as long as all operations on the underlying view is associative. In Figure 1, the operations on the underlying `llist` view type are associative, assuming that `list_monoid` and `llist` are implemented correctly. Hence, upon return from `update_list`, the list `ll` will contain the following: its initial contents at the start of `update_list`; followed by elements inserted by `foo`; followed by the elements inserted by the parallel for loop, that is, the integers `0`, `1`, ..., `n` in order.

The dag model for dynamic multithreading

The execution of a Cilk program can be modeled as a directed acyclic graph, (or dag for short) [5], as shown in Figure 2. In this model, vertices represent ***strands*** — sequences of one or more executed instructions containing no parallel control — and edges denote parallel control dependencies between strands.

The dag unfolds dynamically as the program executes. A function F spawning another function G creates a ***spawn strand*** in the dag, which has two outgoing edges: one leading to the first strand in G, and another leading to a ***continuation strand***, representing the statement after the spawn in F. Executing a sync creates a ***sync strand*** in the dag, which has more than one incoming edge, joining the previously spawned subcomputations.

Figure 2 shows a Cilk program and its corresponding computation dag that we shall use as a running example. The strands in the dag are numbered in their ***serial execution order*** — the depth-first traversal of the dag in which every spawned child is visited before its continuation.

```
23 void foo() {
24   cilk_spawn bar();
25   cilk_spawn baz();
26   quux();
27   cilk_spawn quuz();
28   // ... compute ...
29   cilk_sync;
30 }
31
32 void baz() {
33   cilk_spawn qux();
34   // ... compute ...
35   cilk_sync;
36 }
```

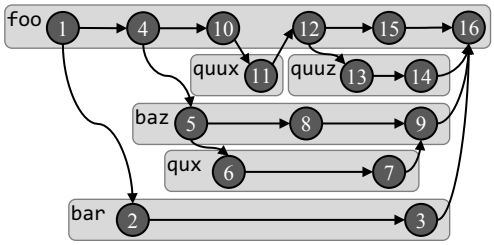

Fig. 2. An example of a Cilk program (left) and its corresponding computation dag (right). In the computation dag, dark circles represent strands, and edges represent parallel control dependencies between strands. The strands are numbered according to their serial execution order. Light rectangles are labeled with the function instantiation and enclose the strands that execute within that instantiation.

For convenience, we shall assume that strands respect boundaries of ***Cilk functions*** — functions that can spawn — meaning that 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 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. Meanwhile strands 9 and 10 are logically in parallel.

Dynamic multithreading via work stealing

Cilk keywords denote the *logical* parallelism of the computation, rather than the actual parallel execution. During execution, as the computation dag unfolds, Cilk's runtime scheduler determines how to best map the strands onto available processors. Specifically, Cilk employs a randomized work-stealing scheduler [5, 20] that dynamically load balances the parallel computation across available ***worker threads*** managed by the runtime in a way that respects the dependencies specified by the keywords. Typically, a worker executes a Cilk computation in its serial execution 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 randomly chooses another worker in the system as its ***victim*** to steal work from. If the chosen victim worker has excess work, the thief may ***steal*** some of this work by resuming the continuation of the stolen function. Notably, a worker's behavior mirrors precisely the behavior of its serial elision between successful steals, and Cilk's support for reducers implements significant optimizations based on this fact.

Cilk's 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 encounters an 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 the

a)

```
37 void update_list(int n, llist<int>& ll) {
38   reducer< list_monoid > ll_reducer;
39
40   ll_reducer.set_value(ll);
41   int x = cilk_spawn foo(n, ll_reducer);
42   cilk_for (int i = 0; i < n; ++i) {
43     ll_reducer.view().insert(i);
44   }
45   ll = ll_reducer.get_value();
46 }
```

b)

```
47 void race(int n, llist<int>& ll) {
48   int length = 0;
49   // SUBTLE BUG: This copy constructor
50   // performs a shallow copy.
51   llist<int> copy(ll);
52   length = cilk_spawn scan_list(ll);
53   update_list(n, copy);
54   cilk_sync;
55   return;
56 }
```

Fig. 3. Code examples that illustrate how races can occur in Cilk programs that use reducer hyperobjects. **a)** A Cilk program that closely resembles the one shown in Figure 1a that contains a view-read race on the access to the `ll_reducer`. This subfigure differs from Figure 1a in that the `cilk_sync` on line 9 is missing. **b)** A Cilk program that calls the `update_list` function shown in Figure 1a in a way that leads to a determinacy race with the reduce operation of `ll_reducer`.

program's serial elision. In the absence of a race, as long as the REDUCE operation is semantically associative, the final resulting view is the same as if the program were run serially.

In the description of the dag model above, we have omitted how one models the execution of view-aware strands. In particular, the model described thus far does not account for the additional instructions generated by the runtime that create new views and reduce views together. Section 6 describes formally how the runtime manages views and how one models the view-aware strands.

3 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 take the code example shown in Figure 1 and show how a small modification to the code can result in a subtle view-read race. We also walk through the example to illustrate how a different subtle programming error can trigger a determinacy race between the user code and a reduce operation on a reducer.

Example of a view-read race

To illustrate how a view-read race can occur, let us consider the code for the `update_list` routine shown in Figure 3a. Note that the only difference between the `update_list` shown in Figure 1a (described in Section 2) and Figure 3a is the lack of `cilk_sync` before the call to `ll_reducer.get_value()`.

The `cilk_sync` statement was there to ensure that, before accessing the final results produced by `ll_reducer`, all previously spawned subcomputations that could update `ll_reducer` have returned. In this case, since `ll_reducer.get_value()` in line 45 is called without the `cilk_sync` statement, `foo` spawned in line 41 — which could update `ll_reducer` — may or may not have returned when control reaches `ll_reducer.get_value()`. Consequently, the final result returned by `ll_reducer.get_value()` can be nondeterministic: it may or may not include updates to `ll_reducer` performed by `foo`.

Example determinacy race involving a reducer

The code in Figure 3b can exhibit a determinacy race between the `race` and the `update_list` routines shown in Figure 1a. In this code, the `race` routine invokes `scan_list`, which iterates through the elements of `llist` 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 53. Because `update_list` might actually insert into the list, the `race` routine makes a copy of the list first at line 51 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 of the user-defined view type `llist` implicitly invoked on line 51 performs only a shallow copy. That means, even though `copy` is a new `llist` object, created with its own distinct head and tail pointers, the two `llist` instances, `ll` and `copy`, still point to the same set of linked-list nodes, leading to a determinacy race in the code.[1] In particular, whenever `scan_list` reaches the last linked-list node originally in `copy` and reads that node's `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` routine 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. The instruction that eventually writes to the `next` pointer and constitutes the race occurs in a REDUCE operation that eventually appends to the original view of `ll_reducer`, as initialized in Figure 1a on line 4. Existing race-detection tools for Cilk programs, such as Cilk Screen [24] or C-RACER [44], will not catch this particular race, because the determinacy race involves a view-aware instruction executed in a REDUCE operation.

4 THE PEER-SET ALGORITHM

This section presents the Peer-Set algorithm for detecting view-read races. We define the "peer-set semantics" that reducers obey in terms of this dag computation model. We use peer-set semantics to formally define based view-read races. We formally define the view-read races based on peer-set semantics, and we describe how the Peer-Set algorithm detects such races. We argue that the Peer-Set algorithm executes in time $O(T\alpha(x, x))$, where T is the serial running time of the Cilk program under test, x is the number of reducer variables in that Cilk program, and α is Tarjan's functional inverse of Ackermann's function.

Peer-set semantics

The views of a Cilk reducer obey ***peer-set semantics*** for dictating which updates are guaranteed to be reflected in those views. To define peer-set semantics formally, let h denote a reducer used in a Cilk program, and let u denote a strand in that Cilk program's execution. The ***peers*** of u, denoted $peers(u) = \{w \in V : w \parallel v\}$, is the set of strands in the Cilk computation that are logically in parallel with u. For example, in Figure 2, strand 1 has no peers, strand 4 only has strands 2 and 3 as its peers, and the peers of strand 5 are $\{2, 3, 10, 11, 12, 13, 14, 15\}$. We define peer-set semantics formally as follows:

Definition 4.1 (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 $h(v) = h(u) \otimes a_1 \otimes a_2 \otimes \ldots \otimes a_k$.

In other words, for two strands in the dag, the updates reflected in the reducer views corresponding to those strands depends on whether those strands have the same peers. For example, in Figure 2, strands 5 and 9 have the same peers, and therefore the view of a reducer at strand 9 will reflect the updates that occurred since strand 5. Strands 10 and 14, meanwhile, do not have the

[1] Such a shallow copy constructor can also lead to subtle memory errors, such as double freeing on some list nodes, when views are automatically destructed by the underlying runtime system.

same peers, because strands 12 and 13 are peers of 14, but not of 10. Hence, the view at strand 14 is not guaranteed to reflect the updates since strand 10.

View-read races

View-read races are formally defined in terms of accesses to a reducer. A ***reducer-read*** operation is an operation to either create the reducer, retrieve its value, or reset its value. Reducer-read operations are distinct from invocations of CREATE-IDENTITY, UPDATE, or REDUCE, which operate on an underlying view of a reducer, rather than the reducer itself.

A view-read race occurs when two strands with different sets of peers both perform reducer-read operations. 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 peers, a view-read race exists between strands 1 and 9. More generally, if a strand v retrieves the value of a reducer h and strand u is the latest strand in the serial execution order before v that created or reset the value of h, then a view-read race exists between strands u and v if $peers(u) \neq peers(v)$.

A Cilk program might behave deterministically, even if it contains a view-read race. For instance, consider the code example in Figure 3a. There is indeed a view-read race between `ll_reducer.set_value(ll)` on line 40 and `ll_reducer.get_value()` on line 45 since they have different sets of peers. In particular, `foo` spawned in line 41 is a peer of the strand performing `ll_reducer.get_value()` but not a peer of the strand performing `ll_reducer.set_value(ll)`. If `foo` does not invoke any UPDATE or reducer-read operations on `ll_reducer`, however, then the `update_list` routine could behave deterministically, rendering the view-read race ***benign***. Nevertheless, the Peer-Set algorithm conservatively declares this situation 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 perform reducer-reads and which strands have the same peers.

During program execution, the Peer-Set algorithm assigns a unique ID to every Cilk function instantiation. For each instantiated Cilk function F, the algorithm maintains a ***shadow frame*** on a ***shadow stack***, which is pushed and popped in synchrony with the function-call stack. The shadow frame for F holds two scalars, $F.ls$ and $F.as$, and three "bags," $F.I$, $F.C$, and $F.O$. Each ***bag*** stores a set of ID's for completed function instantiations in a fast disjoint-set data structure [10, Ch. 21]. The contents of these scalars and bags are defined as follows:

- The ***ancestor-spawn count***, $F.as$, stores the total number of spawns that each ancestor F' of F has performed since F' last synced.
- The ***local-spawn count***, $F.ls$, stores the number of spawns F has executed since F last synced.
- The ***I-bag***, $F.I$, contains the ID's of all completed descendants of F with the same peers as the first (or **i**nitial) strand of F.
- The ***C-bag***, $F.C$, contains the ID's of all completed descendants of F with the same peers as the last **c**ontinuation strand executed in F. If F has not spawned since it last synced, then $F.C$ is empty.
- The ***O-bag***, $F.O$, contains the ID's of all **o**ther completed descendants of F not in $F.I$ or $F.C$.

For each Cilk function F, we refer to the sum of the ancestor-spawn and local-spawn counts, $F.as + F.ls$, as the ***spawn count*** of F. As such, the spawn count of F corresponds to the number of spawn statements executed by F and F's ancestors since each last synced. As an example of ancestor- and local-spawn counts, consider the dag in Figure 2b. In this dag, when strand 8 executes,

When F calls or spawns G:

```
1  if F spawns G
2        F.ls += 1
3        F.O ∪= F.C
4        F.C = ∅
5  G.as = F.as + F.ls
6  G.ls = 0
7  G.I = MakeBag({G})
8  G.C = MakeBag(∅)
9  G.O = MakeBag(∅)
```

When G returns to F:

```
1  F.O ∪= G.O
2  if F spawned G
3        F.O ∪= G.I
4  elseif F.ls = 0
5        F.I ∪= G.I
6  else
7        F.C ∪= G.I
```

When F syncs:

```
1  F.ls = 0
2  F.O ∪= F.C
3  F.C = MakeBag(∅)
```

When F reads reducer h:

```
1  if FindBag(reader(h)) is an O-bag or
           reader(h).s ≠ F.as + F.ls
2        a view-read race exists
3  reader(h) = F
4  reader(h).s = F.as + F.ls
```

Fig. 4. Pseudocode for the Peer-Set algorithm. The MakeBag routine creates a new bag with a specified initial contents. The FindBag routine finds the bag containing the specified element by finding the corresponding set in the disjoint-set data structure.

the ancestor-spawn count for `baz` is 2, because its ancestor, `foo`, spawned at strands 1 and 4 before the execution of strand 8. The local-spawn count for `baz`, meanwhile, is 1, because strand 5 in `baz` spawned before strand 8 executes. Hence the spawn count for strand 8 is 3.

The Peer-Set algorithm also maintains a ***shadow space*** of shared memory, called *reader*, which maps each reducer to two pieces of data: the strand that accessed it last, and the corresponding spawn count. Specifically, for each reducer h, $reader(h)$ stores the ID of the Cilk function F that last read h, while the associated field $reader(h).s$ stores the spawn count of F when it last read h.

Figure 4 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 $F.ls$ 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 $G.O$ is always combined with $F.O$, the bag $G.I$ is combined with $F.I$ only if F has not spawned since it last synced; otherwise $G.I$ is combined with $F.C$. The bag $G.C$ is guaranteed to be empty when G returns to F because Cilk functions implicitly sync before they return.

The following theorem justifies that the Peer-Set algorithm runs in nearly linear time.

Theorem 4.2. *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 functional inverse of Ackermann's function.*

Proof. The pseudocode in Figure 4 shows that, at each point in the program execution, the Peer-Set algorithm performs at most a constant number of operations on bags plus a constant amount of additional work. Furthermore, the shadow space *reader*, maintained by the Peer-Set algorithm, stores only x entries, one for each reducer variable. The theorem thus follows from a similar analysis as that for the SP-bags algorithm [16, Thm. 1]. □

5 CORRECTNESS OF THE PEER-SET ALGORITHM

This section presents a proof that the Peer-Set algorithm correctly detects view-read races, that is, it detects a race if and only if one exists. We sketch the intuition for why the Peer-Set algorithm is correct. To formally argue that the Peer-Set is correct, we describe the representation of a Cilk computation dag as an "SP parse tree," and we show how peer-set semantics can be modeled simply within an SP parse tree. Finally, we argue mathematically for the Peer-Set algorithm's correctness.

Intuition

To understand how the Peer-Set algorithm works, consider its behavior as it executes a Cilk function. We shall use the dag illustrated in Figure 2b as a running example. In particular, we shall examine how bags are maintained.

Consider the contents of the bags associated with a Cilk function G when it returns. Because G is guaranteed to be synced, the pseudocode in Figure 4 shows that the C-bag $G.C$ is empty, the I-bag $G.I$ contains the ID's of all descendants of G with the same peers as the first strand in G, and the bag $G.O$ contains the ID's of all other descendants of G. In the dag in Figure 2b, for example, when `baz` returns, bag `baz`.I contains the ID for `baz`, and bag `baz`.O contains the ID for `qux`.

When G returns to its caller or spawner F, the Peer-Set algorithm merges the contents of the O- and I-bags $G.O$ and $G.I$ into bags in F. Let us see how this merger captures the peer-set relationships of these descendant functions with the strands in F.

Because the functions identified in $G.O$ do not share the same peers as the first strand of G or the last continuation strand executed by G, they must have different peers from any strand in F, regardless of whether G is called or spawned. Therefore, bag $G.O$ is always unioned with $F.O$. In the example dag in Figure 2b, when `baz` returns to `foo`, unioning bag `baz`.O with bag `foo`.O correctly identifies that `qux` has a distinct peer set from every strand in `foo`.

As for bag $G.I$, there are two cases.

Suppose that F spawned G. By definition of a spawn, all descendants of G must have different peers from all strands in F. In particular, the continuation strand in F after the spawn of G is a peer of all descendants of G. The bag $G.I$ is thus unioned with the bag $F.O$ when G returns. In the example dag in Figure 2b, because `foo` spawned `baz` at strand 4, every strand in `baz` is logically in parallel with strand 10. Therefore, the strands in `baz` all have distinct peers from the strands in `foo`.

Now suppose that F called G. If F had no outstanding spawned children, then the first strand in G has the same peers as the first strand in F, and the bag $G.I$ is therefore unioned with $F.I$. 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 has different peers from the first strand in F but the same peers as the last continuation strand executed in F. The bag $G.I$ is therefore unioned with the bag $F.C$, and its contents remain there until F either spawns again or syncs. In the example dag in Figure 2b, strand 11 has different peers from strand 1, but the same peers as strand 10, the caller of `quux`. Thus, when `quux` returns to `foo`, unioning the bag `quux`.I with the bag `foo`.C correctly identifies that strands 10 and 11 have the same peers.

Ideally we'd like to keep only two bags, $F.I$ and $F.O$, for those descendants that have the same peers as F's first strand and for those that do not. But these two bags are insufficient. Consider an example where F spawns off G, reads a reducer, calls some Cilk function, and reads the reducer again. These two reducer-reads share the same peers, but these peers are different from those of F's first strand. To properly handle this scenario, the Peer-Set algorithm keeps the bag $F.C$ as a special placeholder that holds descendants with the same peers as F's last-executed continuation strand.

Now let us consider detecting a view-read race. If a strand v reads a reducer h, and $reader(h)$ is in some ancestor's O-bag, then it certainly has different peers from v. In this case, the Peer-Set

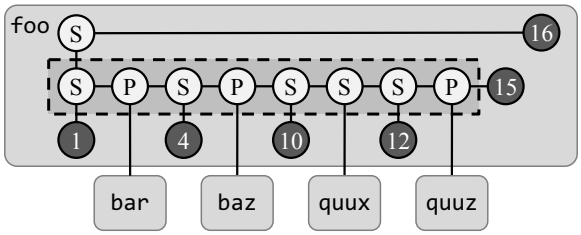

Fig. 5. The canonical SP parse tree for the function instantiation `foo` in the computation dag in Figure 2b. The internal nodes of a sync block are indicated by the darkened rectangle outlined by a dashed line.

algorithm correctly declares a view-read race. If *reader*(h) is in the I-bag for some ancestor F whose first strand is u, then *reader*(h) has the same peers as u. The currently executing strand v thus may or may not have the same peers as u. To handle this case, the Peer-Set algorithm compares the spawn count of v against that of *reader*(h), which is stored in *reader*(h) . s. This spawn count must match the spawn count of u. If v has the same spawn count, then no ancestor of v below F added a peer to v's set of peers that is not a peer of u. Hence, u and v have the same peers. A similar argument holds for the bag $F.C$ and the last continuation strand executed in F.

SP parse trees

To argue formally for the Peer-Set algorithm's correctness, we adopt the representation of a Cilk computation dag as an "SP parse tree" as introduced by Feng and Leiserson [16]. 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 5 illustrates the SP parse tree corresponding to function `foo` in the dag in Figure 2b. The leaves of the SP parse tree are strands in the dag, and each internal node is either an S-node, denoting a series composition of its two children, or a P-node, denoting parallel composition of its two children. The SP parse tree in Figure 5 is a ***canonical*** parse tree [16], 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***. Each sync block is laid out as a chain of S-nodes and P-nodes. For each node in such a chain, the left child is either a strand in F or the root of the canonical parse tree for a subcomputation spawned or called in F. The right child, meanwhile, 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 within F.

Feng and Leiserson prove the following lemma [16, Lemma 4] that shows that the series-parallel relationship between two strands u and v in a Cilk computation is encoded in their ***least common ancestor*** in the tree, denoted LCA(u, v), which is the deepest node in the tree that is a common ancestor of both u and v.

Lemma 5.1. *Let u and v be distinct strands in a Cilk computation dag, and let* LCA(u, v) *be their least common ancestor in the SP parse tree for the dag. Then $u \parallel v$ if and only if* LCA(u, v) *is a P-node.*

Proof of correctness

Peer-set semantics can be modeled simply in terms of the SP parse tree that represents a Cilk computation. The following lemma relates peer-sets to the SP parse tree. In particular, we show

that two strands share the same peers if and only if the path connecting them in the SP parse tree consists solely of S-nodes.

LEMMA 5.2. *Two strands u and v have the same peers, $peers(u) = peers(v)$, if and only if the path connecting u to v in the SP parse tree consists entirely of S-nodes.*

PROOF. We first argue 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 $peers(u) \neq peers(v)$.

($\Rightarrow$) Suppose that $peers(u) = peers(v)$ and 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, $w \nparallel v$, and thus $w \notin peers(v)$. This P-node therefore implies that $peers(u) \neq peers(v)$, contradicting the assumption. A symmetric argument shows that no P-node can exist on the path from $\text{LCA}(u, v)$ to v.

($\Leftarrow$) Suppose that $peers(u) \neq peers(v)$ and the path connecting u and v in the SP parse tree consists entirely of S-nodes. 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 $w \parallel u$ and $w \nparallel v$. By Lemma 5.1, we have that $\text{LCA}(w, u)$ is a P-node and $\text{LCA}(w, v)$ is an S-node. Hence, the nodes $\text{LCA}(w, u)$ and $\text{LCA}(w, v)$ differ, and one can show that $\text{LCA}(u, v)$ 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, contradicting the assumption. The case where $w \notin peers(u)$ and $w \in peers(v)$ is similar. □

We now argue that the Peer-Set algorithm identifies strands that are connected by S-nodes in the SP parse tree, which implies that those strands share the same peers. As in [16], we define the ***procedurification*** function $\mathcal{F}$ as the map from strands and nodes in the SP parse tree to Cilk function invocations.

LEMMA 5.3. *Consider an execution of the Peer-Set algorithm on a Cilk computation. Suppose that strand u executes before strand v, and let $\mathcal{F}$ 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 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.*

(1) The ID for $\mathcal{F}(u)$ belongs to either the I-bag or the C-bag of $\mathcal{F}(a)$ when v executes.

(2) The spawn count for $F(a)$ when u executes equals the spawn count for $\mathcal{F}(v)$ when v executes.

PROOF. ($\Rightarrow$) We first show that if there exists a P-node on the path from u to v in the SP parse tree then one of the conditions is violated. Let b be the first such P-node.

Suppose that b lies on the path between a (inclusive) and u. Then u must lie in a subtree of b. If u is in the left subtree of b, then $\mathcal{F}(b)$ spawned a function F' which is either $\mathcal{F}(u)$ or an ancestor of $\mathcal{F}(u)$. The pseudocode in Figure 4 shows that when F' returns the procedure ID of $\mathcal{F}(u)$ is placed in the O-bag of $\mathcal{F}(b)$, violating condition 1. Suppose instead that u is in the right subtree of b. Then the Peer-Set algorithm's pseudocode in Figure 4 shows that $\mathcal{F}(u)$ is added to either an O- or a C-bag. Moreover, v is not in a subtree of b, because u executes before v in the serial order. Because b is a P-node, the structure of the canonical SP parse tree implies that b lies inside a sync block and all strands in the right subtree of b are in or invoked from the same sync block. Therefore, u and v are in subtrees under different sync blocks in $\mathcal{F}(b)$, which implies that the Peer-Set algorithm must have executed a sync between the time it executed u and the time it executed v. From the pseudocode in Figure 4, $\mathcal{F}(u)$ is placed in an O-bag when this sync executes. Because no action of

the pseudocode moves ID's from an O-bag into either an I-bag or a C-bag, $\mathcal{F}(u)$ always lies in the O-bag of an ancestor of $\mathcal{F}(b)$ after $\mathcal{F}(b)$ returns, violating condition 1.

Suppose instead that b lies on the path between a (exclusive) and v. Then b must lie in the right subtree of a by construction of the canonical SP parse tree. Consequently, $\mathcal{F}(b)$ spawned a child Cilk function invocation F' that it did not sync before v executed. From the pseudocode in Figure 4, $\mathcal{F}(b)$ incremented its local-spawn count when it spawned F'. The spawn count for $\mathcal{F}(v)$ is therefore at least one larger than that of $\mathcal{F}(a)$ when u executed, violating condition 2.

($\Leftarrow$) We now show that if one of the conditions is violated then there exists a P-node on the path in the SP parse tree connecting u and v.

Suppose that the spawn count for $\mathcal{F}(a)$ when u executes does not match that for $\mathcal{F}(v)$ when v executes, meaning that condition 2 is violated. Then either $F'.ls \neq 0$ for some ancestor F' of $\mathcal{F}(v)$ below and including $\mathcal{F}(a)$, or $\mathcal{F}(a).ls$ changed value between executing u and v. In the first case, F' spawned a child computation that it did not sync before v executed. By construction of the SP parse tree, there therefore exists a P-node on the path from a to v. In the second case, if $\mathcal{F}(a).ls$ increased, then before v executed, $\mathcal{F}(a)$ spawned a subcomputation in the same sync block that invoked the subcomputation containing v, and a P-node therefore exists on the path from a to v. Otherwise, $\mathcal{F}(a).ls$ was non-zero when u executed and was reset to 0 between the executions of u and v, implying that $\mathcal{F}(a)$ executed a sync between executing u and v. By construction of the canonical SP parse tree, a P-node therefore lies on the path from a to u.

Now suppose that the ID for $\mathcal{F}(u)$ belongs to an O-bag of $\mathcal{F}(a)$ when v executes, meaning that condition 1 is violated. If $\mathcal{F}(u)$ is in an O-bag, then the pseudocode in Figure 4 shows that $\mathcal{F}(u)$ must have been placed in an O-bag or a C-bag of one of its ancestors F' below and including $\mathcal{F}(a)$. Consequently, the pseudocode implies that $F'.ls$ was nonzero when $\mathcal{F}(u)$ was added to one of its bags, meaning that F' spawned a subcomputation that it did not sync before u executed. If F' is below $\mathcal{F}(a)$, then there must be a P-node on the path from u to a. Otherwise $F' = \mathcal{F}(a)$, and either $\mathcal{F}(a)$ spawned the subcomputation containing u, or $\mathcal{F}(a).ls$ changed value between the executions of u and v. In the first case, a is a P-node. In the second case, the argument above shows that a P-node lies on the path from u to v. □

Combining Lemmas 5.2 and 5.3, we now show that the Peer-Set algorithm correctly detects view-read races.

Theorem 5.4. *The Peer-Set algorithm detects a view-read race in a Cilk computation if and only if a view-read race exists.*

Proof. Let $\mathcal{F}$ be the procedurification function mapping the SP parse tree to Cilk function invocations.

($\Rightarrow$) We first argue that if the Peer-Set algorithm detects a view-read race then one exists. Suppose the Peer-Set algorithm detects a view-read race on a reducer h when executing strand v. Then Figure 4 shows that either $reader(h)$ belongs to an O-bag or the spawn count $reader(h).s$ does not match the spawn count of $\mathcal{F}(v)$, that is, $\mathcal{F}(v).as + \mathcal{F}(v).ls$. Lemma 5.3 therefore implies that a P-node exists on the path connecting u and v in the SP parse tree, meaning that $peers(u) \neq peers(v)$ by Lemma 5.2. Consequently, a view-read race exists.

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

When v executes, suppose that $reader(h) = \mathcal{F}(w)$ for some strand w. If $w = u$, then because $peers(u) \neq peers(v)$, Lemmas 5.2 and 5.3 imply that a view-read race is reported. If $w \neq u$, then

w must have executed after u in the serial order, in order to overwrite $reader(h)$. We must also have that $peers(w) = peers(u)$; otherwise Lemmas 5.2 and 5.3 imply that a view-read race exists between w and u, and that fact that both w and u executed before v contradicts the assumption that strand v is the earliest strand in the serial order for which a view-read race exists on h. Because $peers(u) \neq peers(v)$, we have that $peers(w) \neq peers(v)$, and by Lemmas 5.2 and 5.3, a view-read race is detected. □

6 THE SP+ ALGORITHM

This section presents the SP+ algorithm for detecting determinacy races in Cilk computations that use reducers. A parallel execution of a Cilk program that uses reducers contains view-aware strands, both from calls to update the reducer in the user code and from runtime-invoked Create-Identity and Reduce operations. The SP+ algorithm extends the SP-bags algorithm [16] to handle this additional complexity. We identify the circumstances in which a determinacy race can exist in a computation that uses reducers. We describe intuitively how the SP+ algorithm detects such determinacy races. We argue that the SP+ algorithm detects determinacy races in time $O((T + M\tau)\alpha(v, v))$, where T is the serial running time of the Cilk program under test, M is the number of steals performed on that computation, τ is the worst-case running time of a Create-Identity or Reduce operation, v is the number of memory locations accessed by the computation, and α is Tarjan's functional inverse of Ackermann's function.

To handle the nondeterminism of how the Cilk runtime system manages views of a reducer, the SP+ algorithm takes a steal specification as part of its input. The steal specification dictates which continuations in the computation are stolen, when new reducer views are created, and the partial order in which views are reduced together. The steal specification thereby identifies a particular execution of a Cilk program that uses a reducer. We shall refer to the particular execution of a Cilk program dictated by a steal specification as a ***steal-specified Cilk computation***.

In principle one could imagine that, rather than examine just one steal-specified Cilk computation, the SP+ algorithm might simulate multiple steal specifications simultaneously as it executes a Cilk program. Although there are an exponential number of possible steal specifications, the analysis in Section 9 shows that only a polynomial number of steal specifications are needed to elicit all possible executions of Create-Identity, Update, and Reduce operations. Nevertheless, evaluating multiple steal specifications simultaneously seems to increase the space and time requirements of the algorithm well beyond the requirements to execute a single steal-specified Cilk computation. Hence, the SP+ algorithm examines just one steal specification at a time, allowing the algorithm to be efficient and appropriate for regression testing particular determinacy-race bugs.

To simplify the description, we shall assume that the Cilk program uses a single reducer, and that the Reduce, Create-Identity, and Update methods all execute serial code.[2] As a result, the execution of each of these functions can be modeled as a single strand in the performance dag. As in [28], we shall refer to a strand that arises from the runtime system's implicit execution of Reduce as a ***reduce strand***, and a strand that arises from the runtime's implicit execution of Create-Identity as an ***init strand***.

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]It is generally the case in practice that the Reduce and Create-Identity methods of a reducer execute serial code.

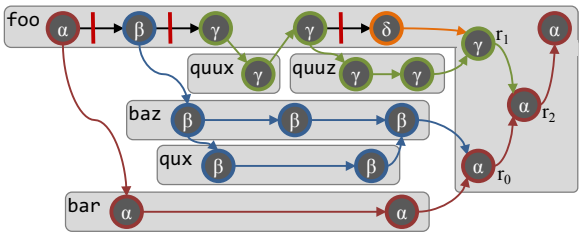

Fig. 6. An example of performance dag, which corresponds to augmenting the user dag in Figure 2 in Section 2 with reduce strands r_0, r_1, and r_2. A vertical bar across an edge indicates that the following continuation strand is stolen. The Greek letter labeling each strand denotes the strand's associated view ID. Strands with the same view ID are highlighted with the same color.

(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, 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 that executes before a sync strand. In particular, if a view $h(w)$ is created at a continuation strand w, and the strand y is the next sync strand after w in the function invocation F containing w, then $h(w)$ is destroyed by a Reduce operation that occurs in F before y [28]. The views of strands entering a sync strand are totally ordered by the dominates relation, and 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 and match the views of all sync strands within the function. This property holds because the first spawned child in a sync block 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 steal-specified Cilk computation can be modeled as a ***performance dag*** [28], which augments the ordinary computation dag model to account for executed 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. In particular, the dependencies among the reduce strands form a ***reduce tree*** before each sync node at which the corresponding call to Reduce occurs. As Leiserson and Schardl describe precisely, these reduce strands form a binary tree whose leaves connect strands with views that are reduced together by the corresponding calls to Reduce operations, and whose root connects to the corresponding sync node. Each reduce strand r whose immediate predecessors have views h_1 and h_2, where $h_1 > h_2$, destroys the dominated view h_2 and inherits the other view, that is, $h(r) = h_1$.

Figure 6 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 `foo`. 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 $\texttt{r}_\emptyset$, for example, reduces the views α and β, destroying β and inheriting the view ID α. Figure 6 also shows how these reduce strands form a reduce tree before the final strand, which is the sync strand in `foo`. 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.

Determinacy races that involve view-aware strands

View-aware strands complicate the circumstances under which a determinacy race occurs. For example, in the performance dag shown in Figure 6, let u be the first strand in function `qux`, and let v be the second strand in function `baz`. Suppose that u and v access the same memory location ℓ with one being a write, and suppose that v is a view-aware strand generated from executing an UPDATE. Because v is a continuation that is not stolen in this execution, the same worker executes v immediately after returning from `qux`. As a result, both u and v observe the same view β of the reducer, as Figure 6 shows. Because v is view-aware, in a different execution in which v is stolen, v would 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, v might not write to ℓ, precluding a determinacy race with u. Because v is view-aware, its logical parallelism with u is not sufficient for it to definitively race with u; it must also operate on a parallel view.

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

- If v is a view-oblivious strand, then a determinacy race exists between u and v if and only if u and v are logically in parallel.
- If v is a view-aware strand, then a determinacy race exists between u and v if and only if u and v are both 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 to detect determinacy races. As it executes, SP+ employs several data structures to keep track of the parallel views created in a steal-specified Cilk computation and to determine the series-parallel relationships between strands, including reduce strands.

Like the SP-bags algorithm, SP+ maintains two 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 function instantiation that last read ℓ, while each location ℓ in *writer* stores the ID for the function instantiation that last wrote ℓ.

For each Cilk function F on the call stack, the SP+ algorithm also maintains a shadow frame containing a set of bags. Each bag sotres a set of ID's for completed procedures in a fast disjoint-set data structure [10, Ch. 21]. 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.

When F spawns or calls G:

1 $G.S = \text{MakeBag}(\{G\}, \text{Top}(F).\,vid)$
2 $p = \text{MakeBag}(\emptyset, \text{Top}(F).\,vid)$
3 $G.P = \langle p \rangle$

When F syncs:

1 $F.S \cup= \text{Top}(F.P)$
2 $p = \text{MakeBag}(\emptyset, F.S.\,vid)$
3 $\text{Top}(F.P) = p$

When spawned G returns to F:

1 $\text{Top}(F.P) \cup= G.S$

When called G returns to F:

1 $F.S \cup= G.S$

When F executes a stolen continuation:

1 $p = \text{MakeBag}(\emptyset,$ **new** view ID$)$
2 $\text{Push}(F.P, p)$

When Reduce R returns to F:

1 $\text{Top}(F.P) \cup= R.S$

When F calls Reduce R:

1 $p = \text{Pop}(F.P)$
2 $\text{Top}(F.P) \cup= p$
3 $R.S = \text{MakeBag}(\{R\}, \text{Top}(F).\,vid)$

Fig. 7. Pseudocode for the SP+ algorithm to maintain bags. Each bag is a set with a *vid* field, which tracks the view ID of that bag. (The view ID of an S-bag matches that of the first P-bag in the P-stack.) This *vid* field is set when the bag is first created and remains invariant as the bag's contents are modified. In particular, when two P-bags are unioned together, the view ID of the destination P-bag is preserved. For a given P-stack x, $\text{Push}(x)$ pushes an element on top of x, and $\text{Pop}(x)$ pops x. $\text{Top}(x)$ reads the topmost bag of x without modifying x. $\text{MakeBag}(S, v)$ creates a new bag with view ID v that contains the elements of S.

- 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 separate P-bags $p \in F.P$ partition this set into subsets based on the parallel views created.

Figures 7 and 8 give the pseudocode of the SP+ algorithm. In particular, Figure 7 gives the pseudocode for maintaining bags, and Figure 8 gives the pseudocode for detecting races. Like the SP-bags algorithm, the SP+ algorithm pushes and pops shadow frames onto a shadow stack in synchrony with the program execution pushing and popping Cilk functions on the call stack.

The SP+ algorithm extends the SP-bags algorithm to additionally push and pop P-bags on a P-stack when reducer views are created or reduced together. 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 of the P-stack. Executing a Reduce operation in F combines the top two P-bags in the P-stack $F.P$, unioning the newer P bag into the older one.

Determinacy races are detected by the code in Figure 8. As the pseudocode shows, different routines are used depending on whether the second strand is view-oblivious or view-aware.

Let us examine how the SP+ algorithm operates by supposing it executes on the computation modeled by the performance dag shown in Figure 6. When it executes the fifth strand u in function `foo`— the stolen continuation labeled with δ — the algorithm pushes a new empty P-bag corresponding to view δ. At this point, the P-stack `foo`$.P$ contains two other P-bags: {`bar`, `baz`, `qux`}, associated with view α, and {`quux`, `quuz`}, 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 u and encounters r_1, the steal specification dictates that the top two P-bags — the empty one representing strand u and the one containing {`quux`, `quuz`} — are unioned before executing r_1. Thus, suppose that r_1, which is a view-aware strand, happens to write to location ℓ last accessed by the first strand in `quuz` labeled γ. Then SP+ will not report a race, because `quuz` belongs to the top P-bag of `foo`

`read` a shared location ℓ by a view-oblivious strand in F:

```
1  if FindBag(writer(ℓ)) is a P-bag
2      a determinacy race exists
3  if FindBag(reader(ℓ)) is an S-bag
4      reader(ℓ) = F
```

`write` a shared location ℓ by a view-oblivious strand in F:

```
1  if FindBag(reader(ℓ)) is a P-bag or FindBag(writer(ℓ)) is a P-bag
2      a determinacy race exists
3  if FindBag(writer(ℓ)) is an S-bag
4      writer(ℓ) = F
```

`read` a shared location ℓ by a view-aware strand in F:

```
1  if FindBag(writer(ℓ)) is a P-bag and FindBag(writer(ℓ)).vid ≠ Top(F.P).vid
2      a determinacy race exists
3  if FindBag(reader(ℓ)) is an S-bag or
         (F is an invocation of Reduce and FindBag(reader(ℓ)).vid == Top(F.P).vid)
4      reader(ℓ) = F
```

`write` a shared location ℓ by a view-aware strand in F:

```
1  if FindBag(reader(ℓ)) is a P-bag and FindBag(reader(ℓ)).vid ≠ Top(F.P).vid
2      a determinacy race exists
3  if FindBag(writer(ℓ)) is a P-bag and FindBag(writer(ℓ)).vid ≠ Top(F.P).vid
4      a determinacy race exists
5  if FindBag(writer(ℓ)) is an S-bag or
         (F is an invocation of Reduce and FindBag(writer(ℓ)).vid == Top(F.P).vid)
6      writer(ℓ) = F
```

Fig. 8. Pseudocode for the SP+ algorithm to detect races. As described in the corresponding pseudocode in Figure 7, each bag is a set with a *vid* field, which tracks the view ID of that bag. For a P-stack x, $\textsc{Top}(F.P)$ reads the topmost P-bag of $F.P$ without modifying $F.P$. $\textsc{FindBag}(f)$ finds the bag that contains f.

when r_1 executes. If the last access of ℓ before r_1 is performed by a strand in `baz`, however, a race will be reported, because `baz` is not the top P-bag of `foo`.

The following theorem and corollary analyze the running time of the SP+ algorithm.

Theorem 6.1. *For a steal-specified Cilk computation A, let $\text{Work}(A_\pi)$ denote the work of the performance dag of A, and let v denote the number of shared memory locations accessed by A. The SP+ algorithm checks A for a determinacy race in $O(\text{Work}(A_\pi) \cdot \alpha(v, v))$ time, where α is Tarjan's functional inverse of Ackermann's function.*

Proof. The pseudocode in Figures 7 and 8 shows that, at each point in the program execution, the SP+ algorithm performs at most a constant number of operations on bags plus a constant amount of additional work. The theorem thus follows from a similar analysis as that for the SP-bags algorithm [16, Thm. 1]. □

Corollary 6.2. *For a steal-specified Cilk computation A, let $T = \text{Work}(A_v)$ denote the work of the user dag of A, let M denote the number of specified steals, let τ denote the worst-case running time of any* Reduce *or* Create-Identity *operation, and let v denote the number of shared memory locations accessed by A. The SP+ algorithm checks A for a determinacy race in $O((T + M\tau)\alpha(v, v))$ time, where α is Tarjan's functional inverse of Ackermann's function.*

Proof. Each specified steal can incur one Create-Identity operation and one Reduce operation, which are not accounted for in the user dag. The work of the performance dag is therefore $O(\text{Work}(A_v) + M\tau)$. The corollary thus follows from Theorem 6.1. □

Intuition for correctness

With respect to detecting races between two view-oblivious strands, it is straightforward to see that SP+ provides the same correctness guarantee as the SP-bags algorithm [16]. Like the SP-bags algorithm, as it executes, SP+ maintains, for every active Cilk function F, two sets of ID's corresponding to F's completed descendants: one set (in the S-bag $F.S$) for those that are logically in series with the currently executing strand, and one set (in a stack of P-bags $F.P$) for those that are logically in parallel with that strand. Both SP-bags and SP+ maintain these sets and use them to detect determinacy races between view-oblivious strands in effectively the same way. SP+ differs only in that it partitions the strands that are logically in parallel across multiple P-bags.

With respect to detecting races between a view-oblivious strand and a view-aware strand, SP+ manages multiple P-bags per Cilk function in order to handle two complications arising from the use of reducers. First, when a view-aware strand is involved, a race between two strands exists only if both the strands are logically in parallel and they operate on parallel views. Consequently, the SP+ algorithm must keep track of the views that strands might operate on. Second, the SP+ algorithm must also keep track of different sets of strands within the a Cilk function F that may end up serialized with some reduce strand executed in F.

SP+ maintains P-bags and their concomitant view ID's in a manner that imitates the Cilk runtime's management of views. Each P-bag has a view ID. When a function F is first spawned or called, it inherits the same view ID as its parent's top P-bag. This behavior reflects how a function called or spawned in Cilk inherits its caller's view of a reducer. Whenever SP+ executes a stolen continuation in F in the steal-specified Cilk computation, it pushes a new P-bag onto the top of $F.P$ with a brand new view ID. This behavior reflects how a new reducer view can be created for each stolen continuation. For a currently executing function F, its top P-bag thus has the view ID corresponding to the view of its currently executing strand. Whenever a Reduce operation occurs in the steal-specified Cilk computation, the SP+ algorithm pops the top P-bag off of $F.P$ and unions it into the next P-bag on top, imitating how the Reduce combines views and destroys the dominated view. Because a necessary set of Reduce operations must occur to destroy all parallel views before a sync, when F executes a sync, SP+ maintains the invariant that only a single P-bag is left in the P-stack for F, which is the same P-bag (with the same view ID) that F had when it started. The view ID effectively simulates how the runtime manages views.

In addition to keeping track of parallel views via view ID's, the multiple P-bags differentiate the sets of strands that can serialize with different Reduce operations. Specifically, whenever a spawned function G returns to F, the ID's corresponding to G's descendants, including G itself, get unioned into F's top P-bag. Each P-bag in $F.P$ thus contains a set of ID's corresponding to F's descendants whose initial strands share the same view. Whenever a Reduce operation occurs, the top two P-bags have the view ID's corresponding to views that the Reduce operation will combine, and the set of ID's they contain correspond to the set of F's descendants that serialize with the reduce operation. Because everything that comes after this reduce strand, including this reduce strand, is in series with the descendants corresponding to the ID's in the top two P-bags, SP+ can safely union them together immediately before the Reduce operation executes.

To detect a potential race with a view-aware strand, SP+ checks that not only are the two strands in parallel, but also that they operate on parallel views. SP+ verifies that the views are parallel by comparing the view ID's of the last access and the currently executing strand. Because the union of the top two P-bags occurs *before* the invocation of the corresponding Reduce operation, any

memory access performed by the reduce strand will have the same view ID's as the descendants in those P-bags. Thus, SP+ achieves the desired effect: the reduce strand is in series with descendants in these two P-bags.

Figure 8 shows 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 that shares the same view as the last access. Let u denote last access in the shadow memory, and let v denote the currently executing strand. By "pseudotransitivity of $\|$" [16, Lemma 7], there is no need to replace u with v in the shadow memory if v is logically in parallel with u, because any strand that comes later in serial order that races with v will race with u as well. SP+ therefore need only to update the last reader or writer if v is in series with u. In the case where v is a reduce strand, however, v is in series with u even if u belongs to a P-bag, as long as the P-bag shares the same view ID.

7 THE SPAWN PARSE TREE AND THE VIEW PARSE TREE

This section introduces the "spawn parse tree" and "view parse tree," which Section 8 uses to formally argue that the SP+ algorithm is correct. For a steal-specified Cilk computation, the spawn parse tree and view parse tree capture the series-parallel relationships between strands and views, respectively, in its performance dag. This section illustrates some example parse trees and shows several useful properties that these parse trees exhibit.

The spawn parse tree and view parse tree together address two complications in showing that the SP+ algorithm correctly detects determinacy races in a steal-specified Cilk computation. First, unlike the SP-bags algorithm [16], the SP+ algorithm must identify when a view-aware strand can modify its view in parallel with another strand, that is, when two strands operate on distinct, parallel views. Second, unlike its user dag the performance dag is not guaranteed to be a series-parallel dag. Hence the logical series-parallel relationships between strands cannot be simply represented using an SP parse tree.

Intuitively, the spawn parse tree maintains the series-parallel relationships among strands in a performance dag, and the view parse tree identifies which strands can operate on their views in parallel. Although this intuition breaks down for reduce strands, we shall see that the spawn and view parse trees together suffice to identify races involving reduce strands as well.

The spawn parse tree

The ***spawn parse tree*** augments the SP parse tree with reduce strands to model nearly all of the series-parallel relationships between strands in a performance dag. A spawn parse tree correctly models all of the series-parallel relationships between non-reduce strands, as well as some of the relationships involving reduce strands. In particular, the spawn parse tree captures the series-parallel relationships between strands based on `cilk_spawn` statements.

For a performance dag A, the spawn parse tree of A is constructed from a modified version of A, called the ***spawn dag***, in which some edges have been replaced to produce a series-parallel dag. For example, Figure 9 illustrates the spawn dag for the performance dag in Figure 6. As Figure 9 illustrates, for each reduce strand r, all but one edge into r is replaced with an edge into the sync strand following r. The one edge into r that is not modified is the edge from the last strand to execute before r in the serial order. Formally, for each reduce strand r in the performance dag, let $u_1, u_2, \ldots, u_k$ be the k predecessors of r in the serial order, and let y denote the nearest sync strand following r. For each of the $k-1$ edges (u, r) where $u = u_1, u_2, \ldots, u_{k-1}$, replace edge (u, r) with an edge (u, y).

The following lemma shows that spawn dag is a series-parallel dag.

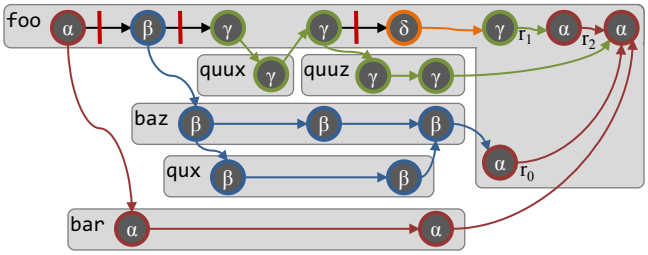

Fig. 9. The spawn dag for the performance dag in Figure 6. Strands are labeled and colored similarly as in Figure 6.

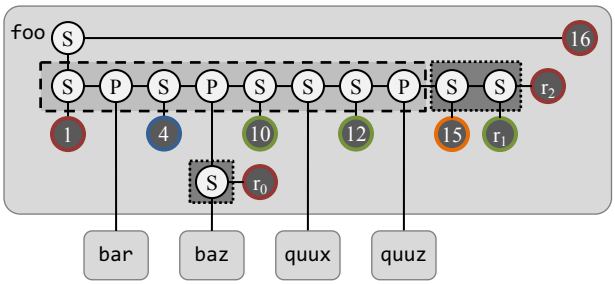

Fig. 10. The spawn parse tree for the performance dag in Figure 6. This spawn parse tree augments the SP parse tree in Figure 5 with reduce chains, which are indicated by the dark rectangles outlined by dotted lines. Strands, which appear at the leaves of the tree, are labeled with either their label in Figure 2 or, for reduce strands, by their label r_0, r_1, or r_2 in Figure 6. Strands with the same view ID are colored similarly, as in Figure 6.

Lemma 7.1. *The spawn dag corresponding to a given steal-specified Cilk computation is a series-parallel dag.*

Proof. To construct a spawn dag recursively using series and parallel compositions, for each function F in the spawn dag, the reduce strands in F before a particular sync strand y are composed in series with F and its spawned subcomputations that sync at y. □

The spawn parse tree of a performance dag A is the canonical SP parse tree for the spawn dag corresponding to A. Figure 10 illustrates the spawn parse tree for the spawn dag in Figure 9. Similarly to how a performance dag augments the ordinary computation dag modeling a Cilk program with reduce and init strands, one can view the spawn parse tree as adding reduce and init strands to the SP parse tree of a Cilk program. For example, compare the spawn parse tree in Figure 10 against the SP parse tree in Figure 5. As this comparison shows, the spawn parse tree appends chains of S-nodes, called ***reduce chains***, to each sync block in its corresponding SP parse tree. These reduce chains connect the existing strands in the SP parse tree to the reduce strands in a manner that reflects their series-parallel relationships in the spawn dag. Furthermore, the edge into each reduce strand r in the spawn dag comes from the last predecessor of r in the serial execution order. Consequently, serial executions of the performance dag and the spawn dag observe all of the same strands in the same order. Hence, a serial execution of a performance dag corresponds to the depth-first, left-to-right traversal of its corresponding spawn parse tree.

The spawn dag captures a subset of the parallel control dependencies between strands in the performance dag, but it misidentifies some pairs of strands as being in parallel with each other.

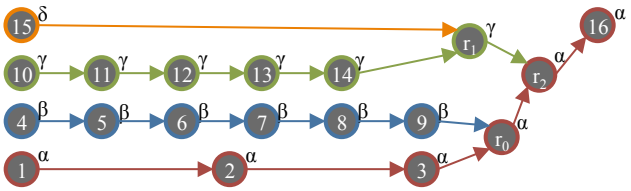

Fig. 11. The view tree for the performance dag in Figure 6. Strands are labeled with either their label in Figure 2 or, for reduce strands, by their label r_0, r_1, or r_2 in Figure 6. Strands are additionally labeled with their view ID and colored to reflect their view ID.

In the spawn parse tree in Figure 10, for example, the least common ancestor between r_1 and any strand in quuz is a P-node, even though the performance dag in Figure 6 shows that r_1 follows the entire execution of quuz.

The following two lemmas show that, except for reduce strands, two strands are logically in parallel in the performance dag if and only if their least-common ancestor in the spawn parse tree, denoted as LCA_S, is a P-node.

Lemma 7.2. *Let u and v be strands in a performance dag, where v follows u in the serial execution order. If $u \parallel v$ then $\text{LCA}_S(u, v)$ is a P-node.*

Proof. The spawn dag is a version of the performance dag where some paths have been removed. Consequently, if $u \parallel v$, then no path connects u to v in either the original performance dag or the spawn dag. The construction of the spawn parse tree from the spawn dag therefore guarantees that $\text{LCA}_S(u, v)$ is a P-node. □

Lemma 7.3. *Let u and v be strands in a performance dag, where v follows u in the serial execution order. If v is not a reduce strand and $\text{LCA}_S(u, v)$ is a P-node, then $u \parallel v$.*

Proof. Compared to the original performance dag, the spawn dag only removes edges that end at reduce strands, replacing an edge that enters reduce strand r with an edge into the nearest sync strand following r in the performance dag. Because v is not a reduce strand, if a path from u to v contained a reduce strand r in the performance dag, then it must also contain the nearest sync strand following r. Therefore, a path from u to v must therefore exist in the spawn dag as well. The lemma therefore follows from the analysis of SP parse trees by Feng and Leiserson, specifically, [16, Lemma 4]. □

For convenience, we show the following property of dags, which generalizes the property shown in [16, Lemma 6].

Lemma 7.4. *Suppose that three strands a, b, and c are encountered in order in a depth-first traversal of a dag G. If a path exists in A from a to b and no path exists from b to c, then no path exists from a to c.*

Proof. Assume for the purpose of contradiction that $v \prec w$. Because $u \prec v$, we have $u \prec w$ by transitivity, contradicting the assumption that $u \parallel w$. □

The view parse tree

The ***view parse tree*** captures which strands in a steal-specified Cilk computation can modify their views in parallel and which strands cannot. The view parse tree is derived from an alternative representation of the performance dag, called the ***view tree***, which is a directed tree that models

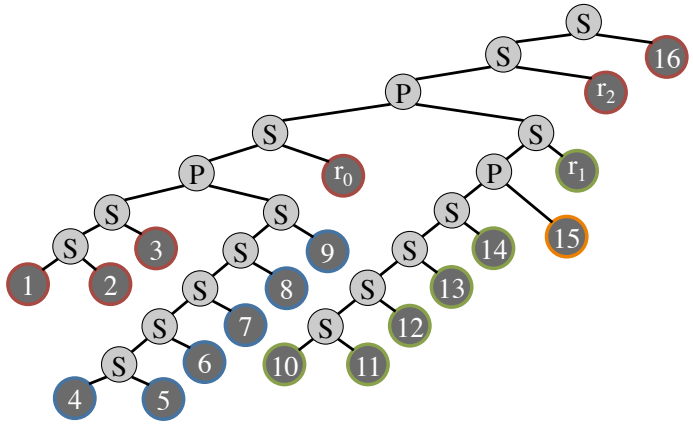

Fig. 12. The view parse tree for the performance dag in Figure 6. Strands, which appear at the leaves of the tree, are labeled with either their label in Figure 2 or, for reduce strands, by their label r_0, r_1, or r_2 in Figure 6. Strands are colored to reflect their view ID.

the series-parallel relationships between views. These series-parallel relationships are implied by the creation of views according to Invariant 2 and the destruction of views by REDUCE operations.

A view tree is constructed from the performance dag of a steal-specified Cilk computation by considering the strands of the dag in their serial execution order and applying the following rules in order:

(1) A strand u for which $h(u)$ is a new, identity view is a leaf in the view tree. By Invariant 2, Strand u is either the first strand in the computation or a continuation strand.
(2) If a strand u has the same view as the strand v immediately before it in the serial execution order — meaning that $h(u) = h(v)$ — then u is the sole successor of v in the view tree.
(3) A reduce strand r that combines the views $h(u)$ and $h(v)$ has incoming edges from u and v in the view tree.

Figure 11 illustrates the view tree for the performance dag in Figure 6. Figure 11 depicts two features of how view trees model reduce strands. First, in Rule 2, the strand u is never a reduce strand, because the view of the strand immediately before a reduce strand r in the serial execution order is always destroyed by r. Furthermore, a reduce strand r always has two predecessors u and v in the view tree, where u and v are the latest strands before r in the serial execution order with the views $h(u)$ and $h(v)$, respectively, that are reduced together by r.

The proof of correctness for SP+ relies on several properties, shown in the following lemma, on the structure of the view tree and its relationship to the performance dag.

LEMMA 7.5. *Consider the view tree of a performance dag, and let G be a subtree of that view tree rooted at a strand v. Let u denote the first strand in G to execute in the serial order of execution. The following properties hold:*

a The views $h(u)$ and $h(v)$ are the same.
b There exists a path in the performance dag from any strand in G to v.
c There exists a path in the performance dag from u to any strand in G.
d Every path in the performance dag that begins at a strand outside of G and ends at a strand inside of G includes u.

Proof. The lemma follows by induction on the construction of the view tree. The case of a single leaf strand, created by Rule 1, and a chain of strands created by repeated applications of Rule 2 are straightforward. We therefore focus on Rule 3.

In the case where a reduce strand r has incoming edges from two strands in the view tree, let G_1 and G_2 denote the two child subtrees of r, where G_1 executes before G_2 in the serial execution order. By construction of the view tree, the strands in G_2 execute immediately after the strands in G_1 in the serial execution order. Let h_1 and h_2 denote the two views reduced together at r, where h_1 and h_2 are adjacent and $h_1 > h_2$. Then the root of G_1 has view h_1 and the root of G_2 has view h_2. We justify each of the properties.

Property a: By induction, the first strand in G_1 has the view h_1. Therefore, r shares the same view as the first strand in the subtree rooted at r.

Property b: Because the roots of G_1 and G_2 both connect to r in the performance dag, by induction, there exists a path from any strand in either G_1 or G_2 to r.

Property c: By induction, h_2 is the view of the first strand x in G_2. Because G_1 and G_2 are not connected in the view tree and all strands with the same view are connected in the view tree, no strand in G_1 shares the view h_2. By Invariant 2, strand x must be the strand where h_2 is created, and therefore, x is a continuation strand of a spawn strand x' in G_1. Because, by induction, there exists a path from u to x', a path from x' to x, and a path from x to any strand in G_2, there must be a path in the performance dag from u to any strand in G.

Property d: By the same argument above, any path in the performance dag into G_2 must therefore include strand x' in G_1. By induction, we thus have that any path from a strand outside of G_1 to a strand in either G_1 or G_2 must include the strand in G_1 that executes first in the serial execution order. □

The view parse tree represents the series-parallel composition of subtrees in the view tree, just as the SP parse tree represents a series-parallel dag. Figure 12 illustrates the view parse tree for the view tree in Figure 11. As Figure 12 shows, a strand u with a single predecessor v in the view tree is composed in series with the subtree rooted at v. If a reduce strand r has two predecessors u and v in the view tree, then the subtrees rooted at u and v are composed in parallel, and r is composed in series with the subtree modeling parallel composition.

The view parse tree thus captures the series-parallel relationships between views of a reducer. Consider two strands u and v in the performance dag, where u executes before v in the serial execution order. Strands u and v can modify their respective views in parallel, denoted $h(u) \parallel h(v)$, only if their least common ancestor in the view parse tree, denoted as $\mathrm{LCA}_V(u, v)$, is a P-node. Otherwise, the value of $h(v)$ reflects updates in the value of $h(u)$, denoted $h(u) \prec h(v)$, and u and v therefore do not operate on parallel views. The following lemma makes this intuition formal.

Lemma 7.6. *For two strands u and v in a performance dag, we have that $h(u) \parallel h(v)$ if and only if $\mathrm{LCA}_V(u, v)$ is a P-node.*

Proof. ($\Rightarrow$) Suppose for the purpose of contradiction that $h(u) \parallel h(v)$ and $\mathrm{LCA}_V(u, v)$ is an S-node a. Without loss of generality, suppose that u executes before v in the serial execution order. Let G_1 and G_2 denote the view subtrees corresponding to the two children of a, where G_1 contains u and G_2 contains v. By construction of the view parse tree, there is a path from u to the root of G_1. Furthermore, because a is an S-node, G_2 contains only v and is composed in series with the root of G_1. Consequently, there exists a path from u to v in the view tree, contradicting the assumption that $h(u) \parallel h(v)$.

($\Leftarrow$) Assume for the purpose of contradiction that $h(u) \prec h(v)$ and $\mathrm{LCA}_V(u, v)$ is a P-node a. Let G_1 and G_2 denote the view subtrees corresponding to the two children of a, where G_1 contains u

and G_2 contains v. Because $h(u) \prec h(v)$, there must exist a path from u to v in the view tree. By construction of the view parse tree, because a is a P-node, G_1 and G_2 must be two parallel subtrees in the view tree whose roots point to a common reduce strand r. There therefore exists a path in the view tree from one subtree of r to the other subtree of r, contradicting the fact that the view tree is a directed tree. □

We conclude this section with four lemmas on the structure of the view parse tree and spawn parse tree for a given performance dag. Section 8 uses these properties to show the correctness of SP+.

Lemma 7.7. *Let u, v, and w be three strands in a performance dag that execute in order in the serial execution order. Suppose that $h(u) \parallel h(v)$ and $h(v) \parallel h(w)$. Then we have $h(u) \parallel h(w)$.*

Proof. In the view parse tree for the performance dag, let $a = \text{LCA}_\text{V}(u, v)$ and $b = \text{LCA}_\text{V}(v, w)$. Lemma 7.6 implies that both a and b are P-nodes. Because u, v, and w execute in order, one can observe that $\text{LCA}_\text{V}(u, w)$ must be one of $\text{LCA}_\text{V}(u, v)$ or $\text{LCA}_\text{V}(v, w)$, and either way, $\text{LCA}_\text{V}(u, w)$ is a P-node. Lemma 7.6 thus implies the lemma. □

Lemma 7.8. *Let u, v, and w be three strands in a performance dag that execute in order in the serial execution order. If $u \prec v$ and $u \parallel w$ and $h(u) \parallel h(w)$, then $h(v) \parallel h(w)$.*

Proof. Assume for the purpose of contradiction that $h(v) \prec h(w)$. If $h(u) \prec h(v)$, then there exists a path in the view tree from v to w, contradicting the fact that $h(u) \parallel h(w)$. If $h(u) \parallel h(v)$, then let w' be the first strand for which $h(w') \prec h(v)$. By construction of the view tree, strand w' is a continuation strand that executes between u and v in the serial execution order, and $h(u) \parallel h(w')$. Lemma 7.5 thus implies that $w' \prec w$ (Property c) and $u \prec w'$ (Property d). Consequently, we have that $u \prec w$, contradicting the fact that $u \parallel w$. □

Lemma 7.9. *For two strands u and v in a performance dag, if $\text{LCA}_\text{S}(u, v)$ is a P-node and $\text{LCA}_\text{V}(u, v)$ is a P-node, then $u \parallel v$.*

Proof. Without loss of generality, assume that v follows u in the serial execution order. Because Lemma 7.3 implies the lemma for non-reduce strands, suppose that v is a reduce strand. Assume for the purpose of contradiction that $u \prec v$ in the performance dag and $\text{LCA}_\text{S}(u, v)$ and $\text{LCA}_\text{V}(u, v)$ are P-nodes. Then Lemma 7.6 implies that $h(u) \parallel h(v)$. Let G denote the subtree of the view tree rooted at v. Because u executes before v in the serial execution order and $h(u) \parallel h(v)$, Lemma 7.5 implies that any path from u to v must contain the strand w, the first strand in G to execute in the serial execution order. Because w is not a reduce strand (by Invariant 2) and $u \prec w$, Lemma 7.3 implies that $\text{LCA}_\text{S}(u, w)$ is an S-node. Furthermore, by construction of the spawn parse tree, reduce strand v occurs in the right subtree of $\text{LCA}_\text{S}(u, w)$, meaning that $\text{LCA}_\text{S}(u, w)$ must be an ancestor $\text{LCA}_\text{S}(w, v)$. Consequently, $\text{LCA}_\text{S}(u, v)$ is an S-node, contradicting our assumption that $\text{LCA}_\text{S}(u, v)$ is a P-node. □

Lemma 7.10. *Suppose that three strands u, v, and w execute in order in the serial order in a performance dag. If $u \parallel v$ and $v \parallel w$ and $h(v) \parallel h(w)$, then $u \parallel w$.*

Proof. If $u \parallel v$ and $v \parallel w$, then Lemma 7.2 implies that both $\text{LCA}_\text{S}(u, v)$ and $\text{LCA}_\text{S}(v, w)$ are P-nodes. One can show that either $\text{LCA}_\text{S}(u, v)$ or $\text{LCA}_\text{S}(v, w)$ is the least common ancestor of u and w in the spawn parse tree, meaning that $\text{LCA}_\text{S}(u, w)$ is a P-node. If w is not a reduce strand, then Lemma 7.3 implies that $u \parallel w$.

Suppose instead that w is a reduce strand, and assume for the purpose of contradiction that $u \prec w$. Because $\text{LCA}_\text{S}(u, w)$ is a P-node, by construction of the spawn dag, $u \prec w$ only when u

is in series with an immediate predecessor of w other than the last immediate predecessor of w to execute in the serial execution order. In the view tree, consider the child subtrees G_1 and G_2 of strand w, where G_1 executes before G_2 in the serial execution order. Because $\text{LCA}_S(u, w)$ is a P-node and $u \prec w$, strand u must be in G_1. Meanwhile, the fact that $h(v) \parallel h(w)$ implies that v is in parallel with both G_1 and G_2 in the view tree. Consequently, v must come either before u or after w in the serial execution order, contradicting the fact that v executes between u and w in the serial execution order. □

8 CORRECTNESS OF THE SP+ ALGORITHM

This section presents a proof that the SP+ algorithm correctly detects determinacy races in a steal-specified Cilk computation. For simplicity, this proof assumes that the Cilk computation operates on a single reducer. It is straightforward to extend the argument to handle more general cases. This section focuses on how SP+ detects races in a single steal-specified Cilk computation. Section 9 discusses how a polynomial number of such SP+ runs can provide the desired coverage for ostensibly deterministic Cilk programs, and Section 10 describes how steal specifications can be given inexpensively.

To formally argue that the SP+ algorithm correctly detects determinacy races, we relate the execution of the SP+ algorithm to the spawn parse tree and view parse tree for a steal-specified Cilk computation. By construction, each Cilk function invocation is represented by an assembly of strands and internal nodes in these parse trees. As in Section 5, we define the ***procedurification*** function $\mathcal{F}$ as the map from strands and nodes in these parse trees to Cilk function invocations.

We start with the following two lemmas that relate the execution of the SP+ algorithm to the structure of the spawn parse tree for a steal-specified Cilk computation. First Lemma 8.1 shows that, when the SP+ algorithm executes a sync instruction in a function F, the P-stack of F consists of just one P-bag. Consequently, the action of the SP+ algorithm to move the contents of that P-bag into the S-bag of F is analogous to the action taken by the SP-bags algorithm at a sync strand [16]. Using this result, Lemma 8.2 then relates the structure of the spawn parse tree to the contents of the S-bags and P-bags that SP+ maintains.

Lemma 8.1. *Consider the execution of SP+ on a steal-specified Cilk computation. When SP+ executes a sync strand in function F, the P-stack $F.P$ contains only a single P-bag.*

Proof. The code for the SP+ algorithm in Figure 7 shows that two operations affect the view ID of the topmost P-bag in $F.P$ or the P-stack of any descendent function of F. First, executing a stolen continuation pushes a new P-bag with a new view ID on top of a P-stack. Second, calling Reduce combines the top two P-bags in a P-stack. Because any view that is created at a continuation strand w in F function is destroyed by a Reduce operation r executed before the sync strand following w, any new P-bag pushed onto $F.P$ when SP+ executes w is popped off of $F.P$ when SP+ subsequently executes r. Hence, when SP+ executes a sync strand in F, the P-stack $F.P$ contains only a single P-bag, namely, the P-bag it contained when F was invoked. □

Lemma 8.2. *Consider the execution of SP+ on a steal-specified Cilk computation. Suppose that strand u executes before strand v, and let $\mathcal{F}$ be the procedurification function mapping the spawn parse tree to Cilk function invocations. Let $a = \text{LCA}_S(u, v)$ be the least common ancestor of u and v in the spawn parse tree.*

- *If a is a P-node, then the procedure ID for $\mathcal{F}(u)$ belongs to a P-bag of $\mathcal{F}(a)$ when v is executed.*
- *If a is an S-node and v is not a reduce strand, then the procedure ID for $\mathcal{F}(u)$ belongs to the S-bag of $\mathcal{F}(a)$ when v is executed.*

- *If a is an S-node and v is a reduce strand, then the procedure ID for $\mathcal{F}(u)$ belongs to either the S-bag or the topmost P-bag in the P-stack of $\mathcal{F}(a)$ when v is executed.*

Proof. The proof extends the argument in [16, Lemma 8], which relates the behavior of the SP-bags algorithm to the SP parse tree for a Cilk computation. We extend this argument to account for the reduce operations and reduce chains in spawn parse trees. We justify that, because the spawn parse tree is the canonical SP parse tree of the spawn dag and the SP+ and SP-bags algorithms are analogous with respect to non-reduce strands, then Lemma 8 in [16] completes the proof.

Suppose that a is in a reduce chain. By construction of the spawn parse tree, a is an S-node, v is a reduce strand in the right subtree of a, and $\mathcal{F}(u)$ is either $\mathcal{F}(a)$, a spawned subcomputation of $\mathcal{F}(a)$, or an invocation of Reduce in the same reduce chain. If $\mathcal{F}(u) = \mathcal{F}(a)$, then the procedure ID for $\mathcal{F}(u)$ is stored in the S-bag $\mathcal{F}(a).S$. Otherwise the pseudocode in Figure 7 shows that when $\mathcal{F}(u)$ returns its procedure ID is placed in the topmost P-bag of the P-stack $\mathcal{F}(a).P$. Because reduce chains consist entirely of S-nodes, no stolen continuation executes in $\mathcal{F}(a)$ between the time that the procedure ID for $\mathcal{F}(u)$ is placed in a P-bag and the execution of v. Consequently, the procedure ID for $\mathcal{F}(u)$ is stored in the topmost P-bag of $\mathcal{F}(a).P$ when v is executed.

Suppose that a is in the spine or a sync block of $\mathcal{F}(a)$. By comparing the pseudocode of the SP+ algorithm in Figure 7 to that of the SP-bags algorithm [16], we observe that the SP+ algorithm moves procedure ID's among S-bags and P-stacks analogously to how the SP-bags algorithm moves procedure ID's among S-bags and P-bags. In particular, Lemma 8.1 implies that when a function F executes a sync strand the topmost P-bag in $F.P$ contains all of the strands in P-bags in $F.P$. The SP+ pseudocode executed at that sync strand therefore merges all procedure ID's in $F.P$ into $F.S$, equivalently to how the SP-bags algorithm moves the contents of its P-bag for F into its S-bag for F. Furthermore, the parent of any reduce chain in the spawn parse tree is a P-node. Therefore, for any strand v that executes after a reduce strand u, if $\mathrm{LCA}_S(u, v)$ is this P-node, then the procedure ID for $\mathcal{F}(u)$ belongs to a P-bag in $\mathcal{F}(a)$ as expected.

Consider the execution of the SP-bags algorithm on the spawn dag of the steal-specified Cilk computation, whose SP parse tree is the spawn parse tree. By induction on the spawn dag, we conclude that, when a is in the spine or a sync block of $\mathcal{F}(a)$, if the SP-bags algorithm stores a procedure ID in an S-bag in $\mathcal{F}(a)$, then the SP+ algorithm stores the same procedure ID in an S-bag in $\mathcal{F}(a)$. Similarly, if the SP-bags algorithm stores a procedure ID in the P-bag in $\mathcal{F}(a)$, then the SP+ algorithm stores the same procedure ID in a P-bag in the P-stack of $\mathcal{F}(a)$. □

Next, the following lemma shows how the view ID's associated with the P-bags maintained by SP+ relate to the structure of the view parse tree.

Lemma 8.3. *Consider the execution of SP+ on a steal-specified Cilk computation. Suppose that strand u executes before strand v. Let $\mathcal{F}$ be the procedurification function mapping the view parse tree to Cilk function invocations, and let $a = \mathrm{LCA}_V(u, v)$ be the least common ancestor of u and v in the view parse tree. Suppose that the procedure ID for $\mathcal{F}(u)$ belongs to a P-bag p in the P-stack $\mathcal{F}(a).P$ when v is executed. If a is an S-node, then $p.vid$, the view ID of p, matches the view ID of the topmost P-bag in $\mathcal{F}(v).P$. Similarly, if a is a P-node, then $p.vid$ does not match the view ID of the topmost P-bag in $\mathcal{F}(v).P$.*

Proof. The code for the SP+ algorithm in Figure 7 shows that, when u is initially added to some P-bag in $\mathcal{F}(a).P$, it is added to the topmost P-bag in $\mathcal{F}(a).P$. Three operations affect the view ID of the topmost P-bag in $\mathcal{F}(a).P$ or the P-stack of any descendent function of $\mathcal{F}(a)$. First, executing a stolen continuation pushes a new P-bag with a new view ID on top of a P-stack. Second, calling Reduce combines the top two P-bags in a P-stack. Finally, calling or spawning a function G from a function F propagates the view ID of the topmost P-bag in G to that in F.

If $a = \text{LCA}_\text{V}(u, v)$ is an S-node, then $h(u) \prec h(v)$, meaning that between executing u and v, for every stolen continuation w that SP+ executes, SP+ executed a REDUCE operation that destroyed the view created at w. Consequently, between the time when u is added to the topmost P-bag in $\mathcal{F}(a).P$ and the execution of v, any P-bag pushed onto a function's P-stack (other than the first P-bag, which is always on the P-stack) is subsequently popped off. The topmost P-bag in $\mathcal{F}(v).P$ therefore has the same view ID as p.

If a is a P-node, then $h(u) \parallel h(v)$, meaning that between executing u and v, SP+ executed a stolen continuation w to create a new view $h(w)$, but SP+ did not execute the REDUCE operation that destroys $h(w)$. Because all views created at continuation strands in a function are destroyed before a function returns, w must have been in a function G on the call stack between $\mathcal{F}(a)$ and $\mathcal{F}(v)$ inclusive. Consequently, the view ID of the topmost P-bag in G's P-stack does not match $p.\mathit{vid}$. Because $\mathcal{F}(v)$ is equal to or a descendant of G, the view ID of the topmost P-bag in $\mathcal{F}(v).P$ also does not match $p.\mathit{vid}$. □

Finally, we combine Lemmas 8.2 and 8.3 with the lemmas in Section 7 concerning the structure of the spawn and view parse trees to prove that the SP+ algorithm is correct.

THEOREM 8.4. *The SP+ algorithm detects a determinacy race in a steal-specified Cilk computation A that uses a reducer if and only if a determinacy race exists.*

PROOF. Suppose that SP+ detects a determinacy race when executing a strand v. If v is a view-oblivious strand, then a determinacy race between v and a strand u that executes before it in the serial execution order requires only that $u \parallel v$. By Lemmas 7.2, 7.3, and 8.2, the SP+ algorithm correctly maintains this logical parallelism relation when v is a view-oblivious strand, and the theorem follows from the argument for the proof of correctness of the SP-bags algorithm [16, Theorem 10].

($\Rightarrow$) Suppose that v is a view-aware strand. Let $\mathcal{F}$ be the procedurification function mapping strands to Cilk function invocations. The pseudocode in Figure 8 shows that one of the following three cases occurs, where TOP gets the top-most element of the given P-stack without modifying that P-stack:

(1) Strand v performs a `write` to location ℓ and *reader*(ℓ) belongs to a P-bag p where $p.\mathit{vid}$ does not equal $\text{TOP}(\mathcal{F}(v).P).\mathit{vid}$.
(2) Strand v performs a `write` to location ℓ and *writer*(ℓ) belongs to a P-bag p where $p.\mathit{vid}$ does not equal $\text{TOP}(\mathcal{F}(v).P).\mathit{vid}$.
(3) Strand v performs a `read` to location ℓ and *writer*(ℓ) belongs to a P-bag p where $p.\mathit{vid}$ does not equal $\text{TOP}(\mathcal{F}(v).P).\mathit{vid}$.

In the first case, suppose that the ID in *reader*(ℓ) is set by a strand u, which executes before v. Because $p.\mathit{vid} \neq \text{TOP}(\mathcal{F}(v).P).\mathit{vid}$, Lemma 8.3 shows that $\text{LCA}_\text{V}(u, v)$ must be a P-node, and therefore Lemma 7.6 implies that $h(u) \parallel h(v)$. If v is not a reduce strand, then because *reader*(ℓ) belongs to a P-bag p, Lemma 8.2 implies that $\text{LCA}_\text{S}(u, v)$ is a P-node. Otherwise, v is a reduce strand, and because *reader*(ℓ) belongs to a P-bag p such that $p.\mathit{vid} \neq \text{TOP}(\mathcal{F}(v).P).\mathit{vid}$, Lemma 8.2 implies that $\text{LCA}_\text{S}(u, v)$ is a P-node. Either way, we have that both $\text{LCA}_\text{S}(u, v)$ and $\text{LCA}_\text{V}(u, v)$ are P-nodes, and therefore Lemma 7.9 implies that $u \parallel v$. Because both $u \parallel v$ and $h(u) \parallel h(v)$, a determinacy race exists between u and v. The other two cases are similar.

($\Leftarrow$) Now suppose that there exists a determinacy race in A on a location ℓ. Let u and v be two strands involved in such a race, where u executes before v and, if there are multiple such determinacy races, we choose the determinacy race for which the second strand executes earliest in the serial order. Suppose again that v is a view-aware strand. By definition of a determinacy race, we have that $u \parallel v$ and $h(u) \parallel h(v)$.

A determinacy race occurs in one of three ways:

(1) Strand u writes ℓ and strand v reads ℓ.
(2) Strand u writes ℓ and strand v writes ℓ.
(3) Strand u reads ℓ and strand v writes ℓ.

In each case, let $\mathcal{F}$ be the procedurification function mapping the spawn and view parse trees to Cilk function invocations. We explicitly prove Case 3. The remaining cases are similar.

Suppose that u reads ℓ and v writes ℓ. When v is executed, let w be the strand such that $\mathit{reader}(\ell)$ stores the procedure ID of $\mathcal{F}(w)$. If $w = u$, then $u \parallel v$ implies that $\mathit{reader}(\ell)$ belongs to a P-bag p (by Lemmas 7.2 and 8.2), and $h(u) \parallel h(v)$ implies that $p.\mathit{vid} \neq \textsc{Top}(\mathcal{F}(v).P.\mathit{vid})$ (by Lemmas 7.6 and 8.3). Consequently, the pseudocode in Figure 8 shows that a determinacy race is reported. If $w \neq u$, then we consider the two cases of whether or not u updates $\mathit{reader}(\ell)$ when it executes.

If u updates $\mathit{reader}(\ell)$, then consider the sequence of updates to $\mathit{reader}(\ell)$ from the time u executes up to and including the time w executes. Let the strands performing the updates be $u_1, u_2, \ldots u_k$, where $u_1 = u$ and $u_k = w$. From the pseudocode in Figures 7 and 8, we have for $i = 1, 2, \ldots, k-1$ that when u_{i+1} executes one of the following two cases applies.

- The procedure ID of $\mathcal{F}(u_i)$ is in an S-bag, in which case Lemmas 8.2 and 7.3 implies that $u_i \prec u_{i+1}$.
- Strang u_{i+1} is a reduce strand, the provedure ID of $\mathcal{F}(u_i)$ is in a P-bag p, and $p.\mathit{vid}$ is equal to $\textsc{Top}(\mathcal{F}(u_{i+1}).P).\mathit{vid}$. In this case, Lemma 8.3 implies that $\mathrm{LCA_V}(u_i, u_{i+1})$ is an S-node, and therefore Lemma 7.9 implies that $u_i \prec u_{i+1}$.

In either case, we have $u_i \prec u_{i+1}$ for $i = 1, 2, \ldots, k-1$, which implies that $u \prec w$ by transitivity. Because $u \parallel v$, Lemma 7.4 implies that $w \parallel v$, and Lemmas 7.2 and 8.2 imply that w is in a P-bag p'. Furthermore, Lemma 7.8 implies that $h(w) \parallel h(v)$, and Lemmas 7.6 and 8.3 imply that $p'.\mathit{vid} \neq \textsc{Top}(\mathcal{F}(v).P).\mathit{vid}$. The pseudocode in Figure 8 therefore shows that a determinacy race is reported.

If u does not update $\mathit{reader}(\ell)$, then when u executes, we must have that the procedure ID of $\mathcal{F}(w)$ equals $\mathit{reader}(\ell)$ for some strand $w \parallel u$ that executes before u. Because $u \parallel v$ and $h(u) \parallel h(v)$, Lemma 7.10 implies that $w \parallel v$, and Lemmas 7.2 and 8.2 imply that w is in a P-bag p. Furthermore, Lemmas 7.4 and 7.7 imply that, regardless of whether $h(w) \prec h(u)$ or $h(w) \parallel h(u)$, we have that $h(w) \parallel h(v)$, and Lemmas 7.6 and 8.3 imply that $p.\mathit{vid} \neq \textsc{Top}(\mathcal{F}(v).P).\mathit{vid}$. The pseudocode in Figure 8 therefore shows that a determinacy race is reported. □

9 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 5), 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 invocations of Update operations. This theorem bounds the number of steal specifications in terms of the ***continuation depth*** of the Cilk computation, which is, overall all points in the serial execution of a Cilk computation, the maximum number of continuations that are not synced at that point. In a Cilk computation, if D is the depth of nested Cilk functions and K is the maximum number of continuations in any sync block, then the continuation depth is bounded by KD. The following theorem considers the Cilk computation's ordinary dag, not its performance dag.

Theorem 9.1. *In a Cilk computation, all possible strands resulting from calls to* Update *can be elicited in* $\Theta(W)$ *steal specifications, where* W *is the continuation depth of the Cilk computation.*

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.

For a strand u in the computation dag, consider the root-to-u path p in the SP parse tree. From the argument above, the value of $h(u)$ 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 $h(u)$ is therefore the number of P-nodes a in p for which u is in the right subtree of a.

A root-to-u path in the canonical SP parse tree passes through at most one sync block in each nested Cilk function F, and each P-node in F on that path corresponds to a continuation on the path to u in that sync block. Consequently, $\Omega(W)$ steal specifications are needed to elicit all possible update strands at the location of u. Because there exists a unique path in the SP parse tree from the root to each strand u, one can choose continuations to steal in a breadth-first manner, where two continuations w_1 and w_2 are stolen in the same specification if the same number of P nodes occur on the root-to-w_1 and root-to-w_2 paths in the tree. Consequently, $O(W)$ steal specifications suffice to elicit all possible strands resulting from calls to Update. □

We now consider the number of steal specifications needed to elicit all possible reduce strands, assuming that the Reduce operation is associative. Because a Reduce operation always combines two adjacent views that have not yet been destroyed, given a sequence $\kappa = \langle k_1, k_2, \ldots, k_K \rangle$ of K views, every Reduce operation on κ can be seen as combining 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 the lower bound that $\Omega(K^3)$ specifications are necessary to elicit every reduce strand.

Theorem 9.2. *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 possible execution of* Reduce *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 that can be produced from combining views in κ corresponds to some subsequence of κ. The ***size*** of h is the length of the subsequence corresponding to h. For example, a reduce strand that reduces the views represented by two 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

$$\begin{aligned} |R| &\geq \sum_{s=n/2+1}^{2(n+1)/3-1} (n - s + 1)(2n - 3s + 2) \\ &= \Omega(n^3) \, . \end{aligned}$$

□

10 RADER

This section presents Rader, our prototype race detector that implements both the Peer-Set and SP+ algorithms. Rader implements both algorithms using compiler-inserted program instrumentation. We evaluated Rader on six benchmarks. When running the Peer-Set algorithm, Rader incurs a geometric-mean multiplicative overhead of 2.56 (with a range of 1.01–6.65) over running the benchmarks without instrumentation. When running the SP+ algorithm, Rader incurs an overhead of 16.94 (with a range of 2.94–47.74). To get a sense of how much of the overhead comes from the instrumentation versus algorithm implementation, we measured the overhead of Rader over running the benchmarks with empty instrumentation, that is, where each instrumented program point calls a function that simply returns. When running the Peer-Set algorithm, Rader incurs a geometric-mean multiplicative overhead of 2.31 (with a range of 1.00–4.64) over running the benchmarks with empty instrumentation. When running the SP+ algorithm, Rader incurs an overhead of 7.93 (with a range of 2.73–24.27). These averages are computed without including overhead for `ferret`, an outlier that has very little overhead, which this section explains.

Implementation

The implementation of Rader consists of three parts. First, Rader contains a library that implements the Peer-Set and SP+ algorithms, as described in Sections 4 and 6. Second, the Rader implementation modifies to the compiler to insert instrumentation that calls into the library. Finally, the Rader implementation modifies the Cilk runtime to execute a Cilk computation serially with steals and calls to REDUCE dictated by a given steal specification.

The Peer-Set and SP+ algorithms each require instrumentation of different program points. Both the Peer-Set and SP+ algorithms instrument program points related to Cilk's parallel control flow, including before and after `cilk_spawn` statements, before and after `cilk_sync` statements, and the entry and exit points to Cilk functions. To instrument these Cilk-related program points, we modified GCC 4.9 to insert instrumentation to identify parallel control constructs in the execution, akin to the Low Overhead Annotations [22] for Intel's Cilk Plus compiler. The SP+ algorithm requires additional instrumentation around memory accesses. For instrumenting memory accesses, we piggyback on the ThreadSanitizer instrumentation [42], which GCC has supported since version 4.8 [21].

To implement the SP+ algorithm, Rader must execute a steal-specified Cilk computation, meaning that it must simulate steals according to an input steal specification. To accomplish this, Rader appropriately modifies, or promotes, various runtime data structures that would be modified if, after

a worker executes the corresponding spawn, the continuation of the parent had been stolen [19]. When the worker resumes the parent later, it acts as if it has stolen 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.

Because 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 have modified the Cilk runtime so that when the worker simulates steals it calls back to Rader to see if it should execute a reduction. Although the modified runtime no longer always performs reductions eagerly, we optimized the steal specifications that Rader uses 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. Let D be the maximum depth of nested spawns, and let K be the maximum number of continuations in any sync block. Because Section 9 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. Hence, Rader only needs to check 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, to check REDUCE operations, or which continuations at a particular depth to steal, to check UPDATE operations. Each steal specification can steal the same continuations in every sync block, and the completeness guarantee still stands, as long as Rader is run with $O(K^3)$ different steal specifications.

In practice, Rader accepts two types of inputs for steal specifications. Rader accepts as an input three values which specify the continuations to be stolen. Alternatively, Rader takes 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 the six benchmark applications described in Figure 13. We converted the pipeline programs `dedup` and `ferret` from the PARSEC benchmark suite [4] to use Cilk linguistics and to write its output using a `reducer_ostream`, an output-stream reducer that is distributed with Intel Cilk Plus [23]. The synthetic `fib` benchmark uses a `reducer_opadd`, which is also distributed with Intel Cilk Plus. All other benchmarks use user-defined reducers. The `pbfs` benchmark uses a Bag unordered-set data structure [28]. The `collision` benchmark uses a hypervector reducer hyperobject for dynamically resizable arrays. The `knapsack` benchmark uses a user-defined struct as a reducer. Rader itself, including the modified runtime, and all benchmarks were compiled with `-O3` optimizations. 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 seven other cores. Each running time is an average over 5 runs.

Figure 14 shows the overhead of Rader when running the Peer-Set algorithm on each benchmark. As the figure shows, compared to the running time of the uninstrumented benchmarks, the Peer-Set algorithm incurs a maximum multiplicative overhead of 6.65 and a geometric-mean multiplicative overhead of 2.56. Compared to running times of the benchmarks with empty instrumentation, meanwhile, the Peer-Set algorithm incurs a similar maximum multiplicative overhead of 4.64 and a geometric-mean multiplicative overhead of 2.31. The overhead of Peer-Set mainly comes

Benchmark	*Input size*	*Description*	*Spawns*	*Syncs*	*Cilk frames*	*mem. acc.*
`collision`	$13K$ pts, $264K$ faces	$3D$ Collision detection	1.67×10^4	7.94×10^4	5.22×10^4	1.97×10^8
`dedup`	*large*	File compression	9.40×10^4	3.00×10^0	9.40×10^4	8.39×10^8
`ferret`	*large*	Image similarity search	2.58×10^2	3.00×10^0	2.59×10^2	3.82×10^4
`fib`	28	Recursive Fibonacci	1.49×10^7	4.48×10^7	4.48×10^7	1.34×10^8
`knapsack`	26	Recursive knapsack	3.53×10^6	3.96×10^6	7.06×10^6	2.37×10^8
`pbfs`	$\|V\| = 2.5M$, $\|E\| = 12.76M$	Breadth-first search	6.23×10^6	6.24×10^6	1.25×10^7	1.33×10^8

Fig. 13. Description of the six benchmarks used to evaluate Rader. Each row describes a benchmark, identified in the *Benchmark* column. The *Input size* column identifies the size of the input provided to each benchmark for the tests. The *Description* column briefly describes each benchmark. The final four columns provide statistics on the execution of each benchmark. The *Spawns* column documents the number of `cilk_spawn` statements executed. The *Syncs* column documents the number of `cilk_sync` statements executed. The *Cilk frames* column documents the number of Cilk frames created, which corresponds to the number of Cilk function instantiations. The *mem. acc.* column documents the number of memory reads and writes that the benchmark performs.

	Benchmark running time (s)		*Overhead of Peer-Set*	
Benchmark	*Uninstrumented*	*Empty instrumentation*	*Uninstrumented*	*Empty instrumentation*
`collision`	0.603	0.607	1.00	1.00
`dedup`	11.596	11.577	1.01	1.01
`ferret`	8.058	8.108	1.01	1.00
`fib`	1.219	1.750	6.65	4.64
`knapsack`	0.459	0.527	3.61	3.15
`pbfs`	0.686	0.697	4.52	4.45

Fig. 14. Rader's overhead when running the Peer-Set algorithm on the six benchmarks. Each row corresponds to a benchmark, identified in the *Benchmark* column. The two columns under *Benchmark running time* give the running times in seconds of the benchmarks with no instrumentation (*Uninstrumented*) and with empty instrumentation (*Empty instrumentation*). The two columns under *Overhead of Peer-Set* give the overhead of running the Peer-Set algorithm compared against the running time of the benchmark with no instrumentation (*Uninstrumented*) and with empty instrumentation (*Empty instrumentation*).

from creating and managing bags in the shadow frames upon spawns and syncs. Consequently, benchmarks with high spawn, sync, and frame counts — such as `fib`, `knapsack`, and `pbfs` — exhibit slightly higher overhead.

Figure 15 shows the overhead of Rader when running the SP+ algorithm on each benchmark. In comparing Figures 14 and 15, we see that the SP+ algorithm incurs substantially more overhead than the Peer-Set algorithm. In particular, compared to the running time of the uninstrumented benchmarks, the SP+ algorithm incurs a maximum multiplicative overhead of 47.74 and a geometric-mean multiplicative overhead of 16.94. Figure 15 shows that approximately a factor of 2–3 of this overhead comes from the instrumentation itself. Compared to the running time of benchmarks with empty instrumentation, the SP+ algorithm incurs a maximum multiplicative overhead of 24.27 and a geometric-mean multiplicative overhead of 7.93.

Figure 15 shows that the overhead of the SP+ algorithm also varies among the benchmarks. The `fib`, `knapsack`, and `pbfs` benchmarks all exhibit high overhead. The `dedup` and `ferret` benchmarks, meanwhile, exhibit very little overhead. The high overheads for `fib`, `knapsack`, and `pbfs` come from the small amount of work each of these benchmarks performs per strand. Moreover, a large

	Benchmark running time (s)		*Overhead of SP+*					
			No steals		*Updates*		*Reduce*	
Benchmark	*Uninst.*	*Empty*	*Uninst.*	*Empty*	*Uninst.*	*Empty*	*Uninst.*	*Empty*
`collision`	0.603	1.437	12.94	5.43	12.89	5.40	13.11	5.50
`dedup`	11.596	12.467	2.94	2.73	2.95	2.74	2.94	2.73
`ferret`	8.058	8.142	1.02	1.00	1.01	1.01	1.01	1.00
`fib`	1.219	3.553	16.99	5.83	16.99	5.80	44.33	15.21
`knapsack`	0.459	1.387	28.84	9.55	32.60	10.79	43.44	14.38
`pbfs`	0.686	1.350	44.22	22.48	44.47	22.60	47.74	24.27

Fig. 15. Rader's overhead when running the SP+ algorithm on the six benchmarks. Each row corresponds to a benchmark, identified in the *Benchmark* column. The two columns under *Benchmark running time* present the running times in seconds of the benchmarks with no instrumentation (*Uninst.*) and with empty instrumentation (*Empty*). The six columns under *Overhead of SP+* present the overhead of SP+ algorithm in three different steal specifications. The columns under *No steals* correspond to runs that use a steal specification in which no continuations are stolen. The columns under *Updates* correspond to runs that use a steal specification in which continuations at half of the maximum continuation depth are stolen. These runs correspond to an execution of SP+ to examine a set of executions of reducer UPDATE operations. The columns under *Reduce* correspond to runs that use randomly chosen steal points, which elicit a subset of possible reductions. For each steal specification, the columns labeled *Uninst.* present the overhead of SP+ compared to the uninstrumented running time of the benchmark, and the columns labeled *Empty* present the overhead of SP+ compared to the running time of the benchmark with empty instrumentation.

part of the work of these benchmarks involves accessing memory — stack memory, specifically, for `fib` and `knapsack` — which also incurs overhead from instrumentation and accessing shadow memory. The `collision` benchmark falls somewhere in the middle, because it performs many fewer spawns and syncs. The low overheads of the `dedup` and `ferret` benchmarks come from various sources. Even though `dedup` performs many memory accesses, its running time is primarily dominated by file I/O. The `ferret` benchmark, meanwhile, is an outlier, both in terms of overhead and the number of instrumented events. It turns out that among all of the library code that comes with PARSEC `ferret` exhibits many determinacy races.[3] Because the reporting of races throws off timing due to printouts, we opt to instrument only the main `ferret` code without the rest of the libraries. As a result, only a small fraction of memory accesses within the computation of `ferret` are instrumented.

Figure 15 illustrates the running time overhead that the SP+ algorithm incurs to handle a steal specification, specifically, to simulate stolen continuations and to create and destroy reducer views as a result. Running the SP+ algorithm on a Cilk computation with an empty steal specification — in which no continuations are stolen — is similar to running the SP-bags algorithm [16] on that Cilk computation. In particular, when the SP+ algorithm uses an empty steal specification, then runs of these two algorithms differ only in that the SP+ algorithm will not report a race when logically parallel memory access occur on the same view. As a result, by comparing the *Updates* and *Reduce* columns against the *No steals* columns in Figure 15, one observes the overhead that SP+ incurs to simulate steals and to check for determinacy races when the computation uses multiple reducer views. As Figure 15 shows, the SP+ algorithm incurs a geometric-mean multiplicative overhead of less than 1.3 to handle non-empty steal specifications. This overhead suggests that the SP+ algorithm incurs minimal additional runtime overhead to implement additional complexity over the SP-bags algorithm.

[3]We separately confirmed that these races exist using Intel's Cilkscreen race detector [24].

Let us look more closely at the overhead for the SP+ algorithm for the three high-overhead benchmarks `fib`, `knapsack`, and `pbfs`. The SP+ algorithm incurs much higher overhead on `fib` and `knapsack` when checking races with randomly chosen steal points (which corresponds to the *Reduce* columns in Figure 15) compared with the other tested steal specifications. The `pbfs` benchmark does not exhibit this behavior, however. This discrepancy comes in part from the small sync-block sizes in both the `fib` and `knapsack` benchmarks — each sync block contains essentially one spawn. Thus, randomly choosing the steal points in each sync block boils down to stealing almost every continuation. Thus the additional overhead incurred per sync block is significant compared to the work in the sync block. Running `pbfs` with randomly chosen steal points, meanwhile, does not exhibit much more overhead compared to the other configurations because `pbfs` has sync blocks with as many as 21 continuations. SP+ with the other configurations does incur a much higher overhead on `pbfs` compared to `fib` and `knapsack`, because of the relatively large memory footprint of `pbfs`. Even though all three benchmarks perform similar numbers of memory accesses, `fib` and `knapsack`, unlike `pbfs`, generally access locations on the stack, which is frequently reused during a serial execution. The use of a shadow memory in SP+ exacerbates the large memory footprint in `pbfs`, and thus SP+ incurs many more cache misses when running `pbfs` compared to running `fib` and `knapsack`.

11 RELATED WORK

Race detection is a rich area actively being studied. Roughly speaking, approaches to race detection can be categorized as either static [1, 2, 6, 14, 31, 37, 46] or dynamic [8, 9, 11, 15, 18, 34, 36, 41, 45, 48]. We focus our discussion on the dynamic approach, which both the Peer-Set and the SP+ algorithms adopt. 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.

Determinacy-race detectors generally use one of two strategies to identify which memory accesses can execute logically in parallel. One strategy uses a labeling scheme to find operations that can execute in parallel. The other strategy, which the Peer-Set and SP+ algorithms adopt, tracks logically parallel tasks within data structures for maintaining disjoint sets.

Researchers have explored a variety of labeling schemes for determinacy-race detection. Nudler and Rudolph [33] proposed an ***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 the labels do not change once they are assigned. As a result, the label size can grow proportionally to the maximum number of fork points in the program, that is, the number of execution points where parallel branches are spawned off. Dinning and Schonberg [13] 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 demonstrate empirically that the task-recycling scheme can be implemented efficiently. Mellor-Crummey [30] 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. 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. Utterback *et al.* [44] developed the WSP-order algorithm, which extens the SP maintenance algorithm of Bender *et al.* to detect determinacy races in parallel in asymptotically optimal running time.

In contrast to the labeling schemes, Feng and Leiserson proposed the SP-bags algorithm [16], 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 it suffices to store only a single reader and a single writer in the shadow memory. Raman *et al.* proposed ESP-bags [38] algorithm, an extension of the SP-bags algorithm that handles the `async` and `finish` constructs in Habanero-Java [7]. They subsequently proposed SPD3 detector [39], also for Habanero-Java, that maintains series-parallel relationships by keeping track of the entire computation tree. The SPD3 detector has a simple implementation and executes in parallel. Both the Peer-Set and SP+ algorithms extend the SP-bags algorithm and similarly use a disjoint-set data structure. As a result, 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 this work and previous work is that the SP+ algorithm handles race detection on computations with reducers, which generate non-series-parallel dags in a nondeterministic fashion. Independently of this work, Dimitrov *et al.* [12] developed a serial algorithm that detects determinacy races on a class of non-series-parallel dags, namely, 2D lattices, by maintaining disjoint trees of nodes within the lattice. Although Dimitrov *et al.* evaluate the soundness and efficiency of their algorithm in theory, they do not discuss its empirical performance. The SP+ algorithm is sound for detecting races in a particular execution, and it is efficient in both theory and practice. This work also shows that a polynomial number of executions of SP+ are sufficient to guarantee complete coverage to accommodate the inherent nondeterminism in the runtime system's management of reducers.

12 CONCLUSION

We have presented the Peer-Set and SP+ algorithms for detecting two unique types of races that arise from incorrect use of Cilk reducer hyperobjects. Both algorithms are correct with respect to a given Cilk program executions. Both are also provably efficient in theory and incur modest overhead in practice. We have shown that for an ostensibly deterministic Cilk program, polynomially many executions of the SP+ algorithm with different steal specifications suffice to provide complete coverage.

Both algorithms execute the Cilk computation under test 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 run serially. An efficient parallel algorithm can therefore lead to a light-weight always-on tool for detecting view-read races. 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 a "left-most" and a "right-most" parallel readers [30]. 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 to achieve parallel speedup for a single execution. Of course, multiple runs of the

SP+ algorithm with distinct steal specifications can be performed in parallel trivially. But achieving the desired parallel running time bound for a single execution, such as for efficient regression testing of determinacy-race bugs, poses greater challenges. 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 a steal specification can cause worker threads to be blocked at certain execution points, which can adversarially affect 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 SPAA 2015 and TOPC reviewers for their excellent feedback.

REFERENCES

[1] Martin Abadi, Cormac Flanagan, and Stephen N. Freund. 2006. Types for Safe Locking: Static Race Detection for Java. *ACM Transactions on Programming Languages and Systems* 28, 2 (March 2006), 207–255.

[2] Rahul Agrawal and Scott D. Stoller. 2004. Type Inference for Parameterized Race-Free Java. In *Verification, Model Checking, and Abstract Interpretation*, Bernhard Steffen and Giorgio Levi (Eds.). Lecture Notes in Computer Science, Vol. 2937. Springer Berlin Heidelberg, 149–160.

[3] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. 2004. On-the-Fly Maintenance of Series-Parallel Relationships in Fork-Join Multithreaded Programs. In *16th Annual ACM Symposium on Parallel Algorithms and Architectures*. 133–144.

[4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*.

[5] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* (1999).

[6] Chandrasekhar Boyapati and Martin Rinard. 2001. A Parameterized Type System for Race-free Java Programs. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*. ACM, Tampa Bay, FL, USA, 56–69.

[7] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2011. Habanero-Java: the new adventures of old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ '11)*. 51–61.

[8] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. 1998. Detecting data races in Cilk programs that use locks. In *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures (SPAA '98)*.

[9] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. 2002. Efficient and Precise Datarace Detection for Multithreaded Object-oriented Programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*. ACM, New York, NY, USA, 258–269.

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (third ed.). The MIT Press.

[11] Joseph Devietti, Benjamin P. Wood, Karin Strauss, Luis Ceze, Dan Grossman, and Shaz Qadeer. 2012. RADISH: Always-on Sound and Complete Race Detection in Software and Hardware. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*. IEEE Computer Society, Portland, Oregon, 201–212.

[12] Dimitar Dimitrov, Martin Vechev, and Vivek Sarkar. 2015. Race Detection in Two Dimensions. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '15)*. ACM, Portland, Oregon, USA, 101–110. DOI: http://dx.doi.org/10.1145/2755573.2755601

[13] Anne Dinning and Edith Schonberg. 1990. 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)*. 1–10.

[14] Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*. ACM, Bolton Landing, NY,

USA, 237–252.
[15] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. 2010. Effective data-race detection for the kernel. In *In Proceedings of the 9th USENIX conference on Operating systems design and implementation.*
[16] Mingdong Feng and Charles E. Leiserson. 1999. Efficient Detection of Determinacy Races in Cilk Programs. *Theory of Computing Systems* (1999).
[17] Jeremy T. Fineman. 2005. *Provably Good Race Detection That Runs in Parallel.* Master's thesis. Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, MA.
[18] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09).* ACM, Dublin, Ireland, 121–133.
[19] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. 2009. Reducers and Other Cilk++ Hyperobjects. In *21st Annual ACM Symposium on Parallelism in Algorithms and Architectures.* 79–90.
[20] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI.*
[21] GCC 4.8. 2014. GCC 4.8 Release Series Changes, New Features, and Fixes. Available at https://gcc.gnu.org/gcc-4.8/changes.html. (2014).
[22] Intel Corporation. 2011. Intrinsics for Low Overhead Tool Annotations. Available from https://www.cilkplus.org/open_specification/intrinsics-low-overhead-tool-annotations-v10. (Nov. 2011).
[23] Intel Corporation. 2013. *Intel® Cilk™ Plus Language Extension Specification, Version 1.1.* Intel Corporation. Document 324396-002US. Available from http://cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_2.htm.
[24] Intel Corporation. 2013. An Introduction to the Cilk Screen Race Detector. https://software.intel.com/en-us/articles/an-introduction-to-the-cilk-screen-race-detector. (April 2013).
[25] I-Ting Angelina Lee, Silas Boyd-Wickizer, Zhiyi Huang, and Charles E. Leiserson. 2010. Using Memory Mapping to Support Cactus Stacks in Work-Stealing Runtime Systems. In *PACT.* ACM, 411–420.
[26] I-Ting Angelina Lee, Aamir Shafi, and Charles E. Leiserson. 2012. Memory-mapping support for reducer hyperobjects. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '12).* 287–297.
[27] Charles E. Leiserson. 2010. The Cilk++ Concurrency Platform. *Journal of Supercomputing* 51, 3 (March 2010), 244–257.
[28] Charles E. Leiserson and Tao B. Schardl. 2010. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *SPAA.*
[29] Don McCrady. 2008. Avoiding Contention using Combinable Objects. Microsoft Developer Network blog post. (Sept. 2008). http://blogs.msdn.com/nativeconcurrency/archive/2008/09/25/avoiding-contention-using-combinable-objects.aspx
[30] John Mellor-Crummey. 1991. On-the-fly Detection of Data Races for Programs with Nested Fork-Join Parallelism. In *Proceedings of Supercomputing'91.* 24–33.
[31] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective Static Race Detection for Java. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06).* ACM, Ottawa, Ontario, Canada, 308–319.
[32] Robert H. B. Netzer and Barton P. Miller. 1992. What are Race Conditions? *ACM Letters on Programming Languages and Systems* 1, 1 (March 1992), 74–88.
[33] Itzhak Nudler and Larry Rudolph. 1986. Tools for the Efficient Development of Efficient Parallel Programs. In *Proceedings of the First Israeli Conference on Computer Systems Engineering.*
[34] Robert O'Callahan and Jong-Deok Choi. 2003. Hybrid Dynamic Data Race Detection. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '03).* ACM, New York, NY, USA, 167–178.
[35] OpenMP Architecture Review Board. 2013. OpenMP Application Program Interface, Version 4.0. Available from http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf. (2013).
[36] Eli Pozniansky and Assaf Schuster. 2007. MultiRace: Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs: Research Articles. *Concurrency and Computation: Practice and Experience* 19, 3 (March 2007), 327–340.
[37] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. 2011. LOCKSMITH: Practical Static Race Detection for C. *ACM Transactions on Programming Languages and Systems* 33, 1, Article 3 (Jan. 2011), 55 pages.
[38] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2010. Efficient Data Race Detection for Async-Finish Parallelism. In *Runtime Verification*, Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann (Eds.). Lecture Notes in Computer Science, Vol. 6418. Springer Berlin / Heidelberg, 368–383.
[39] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2012. Scalable and Precise Dynamic Datarace Detection for Structured Parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming*

Language Design and Implementation (PLDI '12). 531–542.

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

[41] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Race Detector for Multi-Threaded Programs. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP)*.

[42] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA '09)*. ACM, New York, New York, 62–71.

[43] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer. 2009. Phaser Accumulators: a New Reduction Construct for Dynamic Parallelism. In *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*.

[44] Robert Utterback, Kunal Agrawal, Jeremy T. Fineman, and I-Ting Angelina Lee. 2016. Provably Good and Practically Efficient Parallel Race Detection for Fork-Join Programs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '16)*. ACM, New York, NY, USA, 83–94. DOI: http://dx.doi.org/10.1145/2935764.2935801

[45] Christoph von Praun and Thomas R. Gross. 2001. Object Race Detection. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*. ACM, Tampa Bay, FL, USA, 70–82.

[46] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 2007. RELAY: Static Race Detection on Millions of Lines of Code. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE '07)*. ACM, Dubrovnik, Croatia, 205–214.

[47] Martin Wimmer. 2013. Wait-free Hyperobjects for Task-Parallel Programming Systems. In *IPDPS*. 803–812.

[48] Yuan Yu, Tom Rodeheffer, and Wei Chen. 2005. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP '05)*. ACM, New York, NY, USA, 221–234.

Received February 2007; revised March 2009; accepted June 2009