Race Detection and Reachability in Nearly Series-Parallel DAGs

Kunal Agrawal*
kunal@wustl.edu

Joseph Devietti†
devietti@cis.upenn.edu

Jeremy T. Fineman‡
jfineman@cs.georgetown.edu

I-Ting Angelina Lee*
angelee@wustl.edu

Robert Utterback*
robert.utterback@wustl.edu

Changming Xu*
c.xu@wustl.edu

Washington University in St. Louis
†University of Pennsylvania†
‡Georgetown University‡

Abstract

A program is said to have a determinacy race if logically parallel parts of a program access the same memory location and one of the accesses is a write. These races are generally bugs in the program since they lead to non-deterministic program behavior — different schedules of the program can lead to different results. Most prior work on detecting these races focuses on a subclass of programs with series-parallel or nested parallelism.

This paper presents a race-detection algorithm for detecting races in a more general class of programs, namely programs that include arbitrary ordering constraints in additional to the series-parallel constructs. The algorithm performs a serial execution of the program, augmented to detect races, in $O(T_1 + k^2)$ time, where T_1 is the sequential running time of the original program and k is the number of non series-parallel constraints.

The main technical novelty of this paper is a new data structure, R-Sketch, for answering reachability queries in nearly series-parallel (SP) directed acyclic graphs (DAGs). Given as input a graph comprising an n-node series parallel graph and k additional non-SP edges, the total construction time of the data structure is $O(n + k^2)$, and each reachability query can be answered in $O(1)$ time. The data structure is *traversally incremental*, meaning that it supports the insertion of nodes/edges, but only as they are discovered through a graph traversal.

*Supported by NSF Grants CCF-1527692 and CCF-1439062.
†Supported by NSF Grant XPS-1337174.
‡Supported by NSF Grants CCF-1314633 and CCF-1617727.

1 INTRODUCTION

A ***determinacy race*** [20] (or ***general race*** [37]), occurs when two or more logically parallel instructions access the same memory location, and at least one access is a write. Determinacy races can lead to nondeterministic program behaviors, and as such they are often bugs.

Over the years, researchers have proposed several algorithms [6, 20, 21, 36, 41, 42, 48] for performing race detection "on the fly" as the program executes. These race detectors perform a single execution of the program, augmented with a race-detection algorithm, and they generally provide a fairly strong correctness guarantee — for the given input, the race detector reports a race if and only if the program contains a race on that input.

A race detector can be thought of abstractly in the following terms. The valid schedules of a program on a particular input can be modeled by a directed acyclic graph, where nodes correspond to sequential code and edges correspond to dependencies between nodes. Executing the program corresponds to performing a traversal of the dag, where the race detector may choose any valid execution order. Note that the dag is not known *a priori*; rather, nodes are only discovered as the traversal unfolds, i.e., as the program executes. An on-the-fly race detector performs a dag traversal while maintaining two key data structures: (1) An ***access history*** stores a representative set of readers and writers for each memory location; the access history for a location may be updated when executing a node that accesses that location. (2) A ***reachability data structure*** supports queries that determine whether there is a directed path between two already-discovered nodes in the dag. When a node v accesses a memory location ℓ, the race detector performs a reachability query between the node u stored in ℓ's access history and

v. If there is a directed path $u \rightsquigarrow v$, then u is must be scheduled before v in all valid executions and hence they cannot be involved in a race with each other.

The key algorithmic challenge for efficient race detection is in designing an efficient reachability data structure. For an on-the-fly race detector, the reachability data structure must be ***traversally incremental***, meaning that it must support updates as new nodes are discovered by a graph traversal. A traversally incremental data structure may restrict the type of graph traversal supported, which corresponds to restricting the execution order of the program being tested. The most common restriction, adopted in this paper and elsewhere [18, 20], is a depth-first execution order.

In a race-detection application, each memory access may require a query, whereas updates need only be performed on parallel constructs. Thus, optimizing the query cost is more important than the update cost; this paper focuses on supporting constant-time queries.

Most prior work focuses on programs with "nested parallelism", which means that the program can be modeled by a ***series-parallel (SP)*** dag. Restricting the race detector to SP dags greatly simplifies the reachability data structure, giving rise to extremely efficient race detectors with only a constant-factor overhead for both sequential [6, 21] and parallel executions [48] of the program being tested.

Contributions. The key technical contribution of this paper (Section 3) is a new, efficient, traversally incremental reachability data structure, called ***R-Sketch***, for a more general family of graphs. Specifically, R-Sketch applies to a graph formed by taking the union of an n-node series-parallel graph with k arbitrary additional edges. The data structure performs each reachability query in $O(1)$ time and all updates in $O(n+k^2)$ total time.

To put this bound in perspective, in the case that the graph is series parallel ($k = 0$), the SP-order algorithm [6] supports all updates in $O(n)$ time with constant-time queries. If all edges are non-SP edges, i.e., $k = \Omega(n)$, it is possible to maintain the transitive closure directly in $O(k^2)$ with constant-time queries. The bound achieved by R-Sketch is simply the sum of both features; therefore, the bound itself should not be surprising, but achieving it (especially in the traversally incremental setting) is not trivial.

Combining the new data structure with a simple access history (Section 4) yields an efficient on-the-fly race detector for a more general class of parallel programs. In particular, the race detector supports languages with two complementary constructs for expressing parallelism: nested parallelism and arbitrary unstructured synchronization. Nested parallelism is the primary form of parallelism in, e.g., the Cilk family [8, 16, 25, 28, 32], OpenMP tasks [3], Intel's TBB [29], the Habanero family [5, 10], Task Parallel Library [34], X10 [10,12]. Arbitrary unstructured synchronization may be expressed through constructs such as "`put(x)`" and "`await(x)`" (see, e.g., [47]), with an edge in the dag from the `put` to the `await`. This unstructured synchronization can also be used to express other common constructs, such as ***futures*** [4, 24]. Since their proposal in the late 70s, futures have has been incorporated into various parallel platforms [2, 10–12, 23, 26, 31, 35] and have become a popular way to extend nested parallelism. Specifically, detecting races in these programs while executing the program sequentially takes $O(T_1 + k^2)$ time, where k is the number of `put` and `await` calls and T_1 is the sequential runtime of the program — that is, the total asymptotic overhead is an additive $O(k^2)$.

Key Related Work. Related to, but easier than, the problem addressed herein is the static offline problem of building a reachability oracle for a directed graph. For a specific subfamily of n-node planar graphs, which includes SP dags but not SP dags with arbitrary extra edges, Kameda's algorithm [30] builds a reachability oracle in $O(n)$ construction time and supports $O(1)$-time queries. In fact, Nudler and Rudolph's English-Hebrew labeling [38], which is the basis of the traversally incremental SP-order algorithm [6], builds on ideas similar to Kameda's algorithm.

More closely related to our problem, but still static, Wang et al. [51] provide an algorithm, called dual labeling, that supports reachability queries on an n-node directed trees with k non-tree edges added. Dual labeling requires $O(n+k^2)$ space, $O(n+k^3)$ construction time, and answers queries in $O(1)$ time. R-Sketch achieves the same space bound, an improved construction time, generalizes the class of graphs handled (sp-dags instead of trees), and is traversally incremental.

Also related is the problem of labeling each vertex (offline) such that reachability queries can be answered by simply comparing vertex labels. The best practical algorithm we are aware of uses 2-hop labels [15], but its construction time is polynomial. In addition, for graphs with arbitrary edges, no nontrivial bound is known for the label size (which is related to the query time).

The only race detection algorithm that we are aware of for nearly series-parallel programs with the addition of arbitrary dependencies [46] has higher overheads (multiplicative in the number of number of arbitrary edges) — the running time is $O(T_1(k+1)(g+1))$, where k is the number of `await` calls, g is the number of `put` calls and T_1 is the sequential runtime of the program.Therefore, the overhead in this case is multiplicative as opposed to the additive overhead provided by R-Sketch. Some other race detectors, such as FastTrack [22], do not differentiate between nested parallelism and the unstructured parallelism. For a program with sequential running time T_1, n nested parallel constructs, and k arbitrary additional edges, FastTrack achieves a running time of $O(T_1 + (n+k)^2)$. R-Sketch would perform similar to FastTrack's if $k = \Omega(n)$, but is much better if most of the parallelism in the program can be expressed through nested parallelism.

Brief Overview of the Approach. If nodes u and v are related by a directed path, any such path can be characterized in three ways: (1) the path comprises only edges in the SP graph, (2) the path comprises only non-SP edges, or (3) the path includes both SP and non-SP edges. Paths of the first type can be handled efficiently by using an efficient reachability data structure on the series-parallel base graph. The second type can also be addressed simply (though less efficiently) by using a general reachability data structure on the subgraph consisting of just the non-SP edges, e.g., maintaining the transitive closure explicitly. The challenge is in capturing relationships through paths that include both SP and non-SP edges. The algorithm for SP dags would not apply if including the non-SP edges, and a general reachability data structure would be too expensive for SP edges.

Roughly speaking, R-Sketch essentially removes or contracts regions of the series-parallel graph that are irrelevant for paths consisting of both SP and non-SP edges. If there are k non-SP edges, R-Sketch contracts the original n-node graph to a graph of $O(k)$ important nodes. First, for intuition's sake, consider the static case where the dag is known. The rough idea is as follows: (1) omit any recursive series-parallel subdags that lack any non-SP edges, and (2) contract any non-branching chains that remain. Section 3.1 discusses the structural properties that would ensure this static algorithm operates correctly.

Using the ideas from the static data structure to build a traversally incremental data structure, however, adds significant complexity. The challenge is that we do not know if a subdag will contain a non-SP edge until traversing far enough, so we cannot immediately be sure whether a node should be contracted, omitted entirely, or kept. R-Sketch exploits indirection as well as some careful amortization.

Roadmap. The rest of the paper is organized as follows. Section 2 reviews and introduces terminology used throughout the paper. Section 3 presents our reachability data structure. Section 4 discusses the issues relating to the modeling of parallel programs and race detection. Finally, Section 5 overviews related work, and Section 6 draws concluding remarks.

2 PRELIMINARIES

This section reviews series-parallel graphs, terminology, and notation used throughout the paper.

Computations which use only nested parallelism can be modeled as a special class of dags referred to as the ***series-parallel dag (SP-dag)*** [49] that has a single ***source*** node with no incoming edges and a single ***sink*** node with no out-going edges, and can be constructed recursively as follows.

- ***Base Case***: the dag consists of a single node that is both the source and the sink.
- ***Series Composition***: let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be SP-dags on distinct vertices. Then the graph G formed by taking the union of the two graphs, with one additional edge from $sink(G_1)$ to $source(G_2)$, is also series parallel. Moreover, G has source and sink $source(G) = source(G_1)$ and $sink(G) = sink(G_2)$.
- ***Parallel Composition***: let $G_L = (V_L, E_L)$ and $G_R = (V_R, E_R)$ be SP-dags on distinct vertices. Then the following graph is also series parallel: the graph G formed by the union of G_L, G_R, a fork node f with edges from f to both sources, and a join node j with edges from both sinks to j. $source(G) = f$ and $sink(G) = j$. We refer to G_L and G_R as the ***left subdag*** and ***right subdag***, respectively, of both the fork f and join j

This definition of series-parallel dags is one that has *binary forking*, meaning that the out-degree (or in-degree) or each fork (or join) node is exactly two. There are other, more general, definitions of SP dags that allow for higher degree. We use this definition for ease of exposition and without loss of generality — any higher-degree SP dag can be transformed into one with binary forking without asymptotically increasing the size of the graph by simply replacing each fork or join by a tree of forks or joins.

Notation and Other Definitions. We consider reachability on a graph $G = (V, E_{SP} \cup E_{non})$, where G is formed by taking a series-parallel graph $G_{SP} = (V, E_{SP})$ and adding an arbitrary set E_{non} of edges, which we call the ***non-SP edges***. Such a graph would be said to have $k = |E_{non}|$ arbitrary edges added to it. Without loss of generality, we assume that non-SP edges are not incident on fork or join nodes. We further assume that the source node of the entire graph is not a fork node. We assume throughout that the graph description specifies which edges are part of E_{SP} and which are the extra edges E_{non}.[1]

We write $u \prec_G v$ to denote the presence of a directed path from u to v in G and say that u is a ***predecessor*** of v and v is a ***successor*** of u. (The path can be empty, i.e., we always have $v \prec v$.) We use $u \overset{G}{\rightsquigarrow} v$ to refer to a (possibly empty) directed path from u to v in G. We often omit the subscript and superscript when we refer to the entire graph G. In addition, we use $u \prec_{SP} v$ as a short hand for $u \prec_{G_{SP}} v$ and $u \overset{SP}{\rightsquigarrow} v$ as a shorthand for $u \overset{G_{SP}}{\rightsquigarrow} v$.

Consider a node v. If x and y are both distinct predecessors of v and $x \prec y$, then we say that y is a ***nearer*** predecessor

[1] This assumption is realistic for on-the-fly race detection while executing the program — the different edge types would be generated by different linguistic keywords.

of v. Similarly, if x and y are both distinct successors of v, and $x \prec y$, then we say that x is a ***nearer*** successor. When we use the term "nearer" in the next section, we always mean with respect to the series-parallel graph G_{SP}.

3 INCREMENTAL REACHABILITY IN NEARLY SERIES-PARALLEL GRAPHS

This section presents a traversally incremental data structure, R-Sketch, to answer reachability queries in nearly series-parallel (SP) dags. More precisely, we consider an SP dag into which k arbitrary edges have been added. The algorithm builds a data structure to answer reachability queries in $O(n+k^2)$ total time, where n is the number of vertices. Queries themselves can be answered in $O(1)$ time per query.

We are not aware of a data structure that has the same bounds even in the offline setting, but we consider here a particular online setting consistent with executing a parallel program. In the online setting, the dag is fixed *a priori*, but the algorithm is only initially aware of the first node in the dag. The algorithm may choose to execute any node with no unexecuted predecessors. When a node is executed, its outgoing edges as well as the target vertices are revealed.

3.1 Overview of R-Sketch.

We first provide the overview of R-Sketch along with the key properties that guide the design. This description directly suggests a family of static (offline) data structures that answer queries in $O(1)$-time. Section 3.3 describes how to efficiently build such a data structure incrementally as the graph is executed. For now, the reader can interpret the discussion herein as applying to the static *a posteriori* graph. (It should be a straightforward exercise to design an efficient algorithm for the static case, but handling the incremental setting is harder.)

R-Sketch consists of two graphs accompanied by data structures for answering reachability queries on those graphs. The first graph is the series-parallel graph $G_{SP} = (V, E_{SP})$. The second is an auxiliary graph which we denote by $\mathcal{R} = (V_{\mathcal{R}}, E_{\mathcal{R}})$. Note that $\mathcal{R}$ is not series parallel, so it is more expensive to build data structures that answer reachability queries on $\mathcal{R}$.

At a high-level, $\mathcal{R}$ is used to query on reachability between two nodes u and v when $u \overset{G}{\rightsquigarrow} v$ involves non-SP edges. On the other hand, G_{SP} is used to query on reachability between two nodes when the path contains strictly SP edges. Since prior work has addressed how to maintain a reachability data structure on G_{SP} efficiently (see Lemma 3.1), this section focuses on the properties of $\mathcal{R}$ and how to maintain $\mathcal{R}$ on-the-fly. For now, treat both reachability structures as a black box, with the following performance characteristics.

LEMMA 3.1. *There exists a data structure that supports reachability queries on a series-parallel $G_{SP} = (V, E_{SP})$ with the following performance characteristics: the data structure can be constructed in $O(|V|)$ total time, and it supports queries in $O(1)$ time.*

Moreover, the data structure can be built traversally incrementally, with the same total construction time, for any valid execution order of the graph G_{SP}.

Proof. The SP-order data structure [6] is one example of such a data structure. (SP-order is overkill for the static case, but it also applies to the traversally incremental setting.)

LEMMA 3.2. *There exists a data structure that supports reachability queries on a general graph $\mathcal{R} = (V_{\mathcal{R}}, E_{\mathcal{R}})$ with the following performance characteristics: the data structure can be constructed in time $O(|V_{\mathcal{R}}| \cdot |E_{\mathcal{R}}|)$ time, and it supports queries in $O(1)$ time.*

Moreover, the data structure can be built traversally incrementally, with the same total construction time, for any arbitrary ordering of vertex and edge insertions.

Proof. The offline algorithm trivially builds the transitive closure by performing a graph search from each node.

It is nearly trivial to extend this to a traversally incremental algorithm by storing at each vertex the set of predecessors. When adding an edge (u, v), check if v has gained any new predecessors by comparing u and v's predecessor sets. Whenever a vertex v gains a new predecessor (which can happen at most $|V_{\mathcal{R}}|$ times), update the predecessors for all of v's neighbors.

It remains to specify $\mathcal{R} = (V_{\mathcal{R}}, E_{\mathcal{R}})$ and explain how queries on G can be implemented with respect to G_{SP} and $\mathcal{R}$. Since the reachability structure on $\mathcal{R}$ is expensive to construct, the main challenge is developing a solution that correctly answers queries while keeping $\mathcal{R}$ small.

Note that any data structure with a different query/update tradeoff can be substituted as a black box instead of Lemma 3.2, which would also directly impact the running-time of the algorithm.

Properties of the Auxiliary Graph. Here we describe the key features of the auxiliary graph $\mathcal{R}$. We later consider the specifics in more detail, but these properties are enough to imply correctness of the query. Moreover, these properties are the core motivations of algorithm design (both static and incremental).

The vertices in $\mathcal{R}$ are a subset of nodes from the original graph. We call these nodes $V_{\mathcal{R}} \subseteq V$ the ***anchor nodes***. The set of anchor nodes comprises all nodes incident on the non-SP edges E_{non}, as well as some (but not too many) other nodes. The graph $\mathcal{R}$ is designed to ensure the following three properties, the third motivating the extra anchor nodes:

PROPERTY 3.1. *For any two anchor nodes $u, v \in V_{\mathcal{R}}$, we have $u \prec_{\mathcal{R}} v$ if and only if $u \prec_G v$.*

PROPERTY 3.2. $|V_{\mathcal{R}}| = O(|E_{non}|)$ *and* $|E_{\mathcal{R}}| = \Theta(|E_{non}|)$.

Each node v in G is associated with two specific anchor nodes (possibly null): an ***anchor predecessor*** and ***anchor successor***, denoted $PredAnchor(v)$ and $SuccAnchor(v)$, respectively. The key property of these anchor predecessor and successors is as follows.

PROPERTY 3.3. *If not null, the anchor predecessor (or successor) is a predecessor (or successor) in G_{SP}, i.e., for every node v there exists a path $PredAnchor(v) \overset{SP}{\rightsquigarrow} v \overset{SP}{\rightsquigarrow} SuccAnchor(v)$.*

Consider any non-SP edge $(x,y) \in E_{non}$. If there is a path $x \to y \overset{SP}{\rightsquigarrow} v$, then $PredAnchor(v)$ is not null and there exists a path of the form $x \to y \overset{SP}{\rightsquigarrow} PredAnchor(v) \overset{SP}{\rightsquigarrow} v$. Similarly, a path $v \overset{SP}{\rightsquigarrow} x \to y$ implies a path $v \overset{SP}{\rightsquigarrow} SuccAnchor(v) \overset{SP}{\rightsquigarrow} x \to y$.

In other words, $PredAnchor(v)$ is nearer than all other predecessors that have incoming non-SP edges. (It may be null if there are no predecessors with incoming non-SP edges.) Similarly, $SuccAnchor(v)$ is nearer than all other successors having outgoing non-SP edges. (It is null if there are no successors with outgoing non-SP edges).

Reachability Queries. Assuming an auxiliary graph with the aforementioned properties, querying whether $u \prec_G v$ operates as follows:

PRECEDES(u,v)

if $u \prec_{SP} v$ // Query the reachability structure on G_{SP}
    return TRUE
elseif $SuccAnchor(u) \neq$ NULL,
        $PredAnchor(v) \neq$ NULL, and
        $SuccAnchor(u) \prec_{\mathcal{R}} PredAnchor(v)$
        // Query the reachability structure on $\mathcal{R}$
    return TRUE
else return FALSE

Note that the reachability structure on $\mathcal{R}$ is queried only if $u \not\prec_{SP} v$ — this is important for correctness; the second query may return the wrong answer if $u \prec_{SP} v$.

The following lemma says that the query is correct.

LEMMA 3.3. *Assuming Properties 3.1 and 3.3, the query algorithm correctly returns $u \prec v$ if and only if there is a path from u to v in G.*

Proof. There are three cases, corresponding to each of the returns. The first case is that $u \prec_{SP} v$. Then the algorithm trivially returns the correct answer (true).

The second case is that $u \not\prec_{SP} v$ and $u \prec_G v$. Let $p = u \overset{G}{\rightsquigarrow} v$ be any path from u to v. Since there is no path in G_{SP}, p must use at least one non-SP arc. If there is just one non-SP arc (a,y) on the path, then we can rewrite the path as $u \overset{SP}{\rightsquigarrow} a \xrightarrow{E_{non}} y \overset{SP}{\rightsquigarrow} v$. Otherwise, let (a,b) be the first non-SP arc on the path, and let (x,y) be the last non-SP arc on the path. Then $p = u \overset{SP}{\rightsquigarrow} a \xrightarrow{E_{non}} b \overset{G}{\rightsquigarrow} x \xrightarrow{E_{non}} y \overset{SP}{\rightsquigarrow} v$. In either case, since a has an outgoing non-SP arc, we can apply Property 3.3 to conclude that the path $u \overset{SP}{\rightsquigarrow} SuccAnchor(u) \overset{SP}{\rightsquigarrow} a$. Similarly, there also exists a path $y \overset{SP}{\rightsquigarrow} PredAnchor(v) \overset{SP}{\rightsquigarrow} v$. Splicing these paths together appropriately with p, we conclude that there exists a path $p' = u \overset{SP}{\rightsquigarrow} SuccAnchor(u) \overset{G}{\rightsquigarrow} PredAnchor(v) \overset{SP}{\rightsquigarrow} v$. Importantly, there is a path $SuccAnchor(u) \overset{G}{\rightsquigarrow} PredAnchor(v)$. Thus, by Property 3.1 $SuccAnchor(u) \prec_{\mathcal{R}} PredAnchor(v)$ and the query correctly returns true.

Finally, suppose $u \not\prec_G v$. We claim that $SuccAnchor(u) \not\prec_G PredAnchor(v)$, and hence by Property 3.1 $SuccAnchor(u) \not\prec_{\mathcal{R}} PredAnchor(v)$ and the query correctly returns false. We justify the claim by contradiction—if there exists a path $SuccAnchor(u) \rightsquigarrow PredAnchor(v)$, then by Property 3.3 there also exists a path $u \rightsquigarrow SuccAnchor(u) \rightsquigarrow PredAnchor(v) \rightsquigarrow v$, and hence $u \prec_G v$, which is a contradiction.

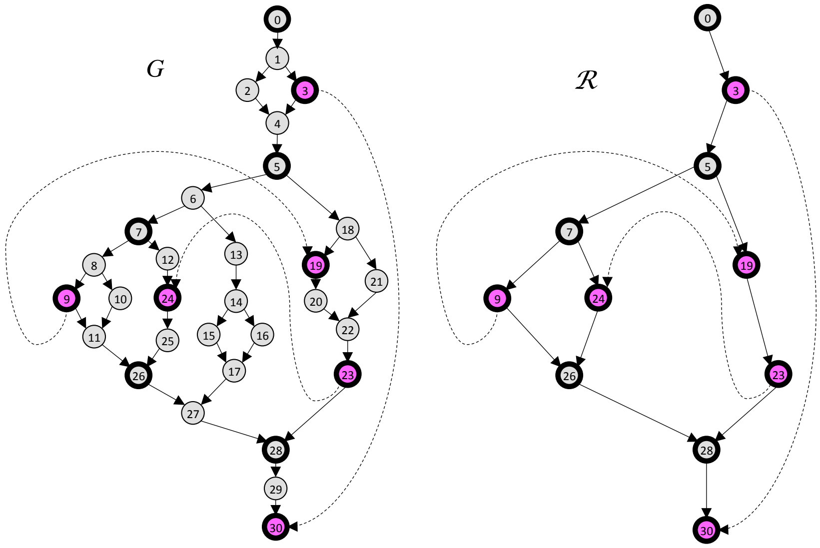
The Auxiliary Graph. We now discuss the *a posteriori* auxiliary graph. In reality, this graph is built-up incrementally while executing G. But since nodes are never removed, it is helpful to reason about the auxiliary graph as a static entity. An example graph and auxiliary graph are shown in Figure 1.

As already noted, $V_{\mathcal{R}}$ includes all of the vertices in G that are incident on non-SP edges. Also included in $V_{\mathcal{R}}$ are just enough fork and join nodes to make Property 3.3 feasible. In particular, the anchor nodes include all of the following:

- all nodes incident on non-SP edges, which we call ***principle anchors***,
- the first node $source(G_{SP})$ of the entire graph (to remove some corner cases),
- all fork nodes whose left and right subdags in G_{SP} each contain at least one principle anchor, and
- all join nodes whose left and right subdags in G_{SP} each contain at least one principle anchor.

In the example, fork node 7 and corresponding join node 26 are both anchor nodes because each of the left and right subdags contain a principle anchor, namely 9 and 24, respectively. In contrast, fork 6 and corresponding join 27 are not anchor nodes because only the left subdag contains any principle anchors.

Note that which nodes are anchors depends only on the series-parallel graph G_{SP} and which nodes have incident non-SP edges. How the nodes are related by non-SP edges



node	anchor pred.	anchor succ.
0	0	0
1	0	3
2	0	5
3	3	3
4	3	5
5	5	5
6	5	7
7	7	7
8	7	9
9	9	9
10	7	26
11	9	26
12	7	24
13	5	28
14	5	28
15	5	28
16	5	28
17	5	28
18	5	19
19	19	19
20	19	23
21	5	23
22	19	23
23	23	23
24	24	24
25	24	26
26	26	26
27	26	28
28	28	28
29	28	30
30	30	30

Figure 1: Example graph G (left), auxiliary graph $\mathcal{R}$ (middle), and the corresponding anchor predecessors/successors (right). The dashed arrows correspond to the non-SP edges E_{non}; omitting the dashed (non-SP) edges from the graphs yields the series-parallel subgraphs G_{SP} and $\mathcal{R}_{SP}$, respectively. Nodes with thicker borders are the anchor nodes, and the magenta nodes are the principle anchors (those incident on non-SP edges). The nodes are numbered by their execution order.

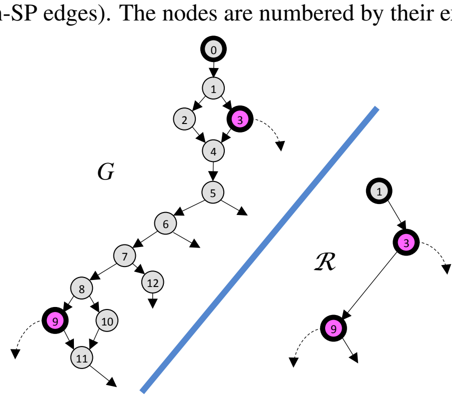


Figure 2: A partial execution of the dag from Figure 1 just after node 12 has been executed. Only nodes that have been processed are displayed.

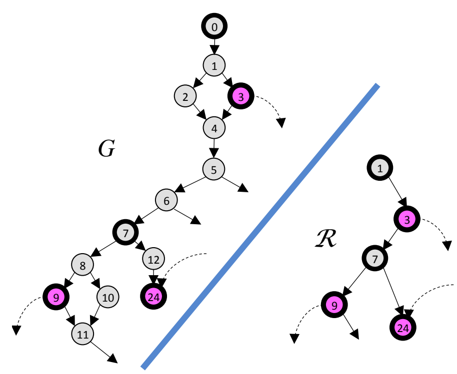


Figure 3: A partial execution of the dag from Figure 1 just after processing node 24. Note that 24 cannot execute because it has an unsatisfied incoming non-SP edge, so the node now blocks and the execution would continue with 6's other child.

as well as the direction of the edges does not affect anything except specific edges in $\mathcal{R}$.

The reason for making certain join nodes anchors is to make it possible to define anchor predecessors consistent with Property 3.3. Otherwise, node 29 could not possibly have an anchor predecessor, as there is no anchor node that comes after both 19 and 24. Similarly, certain forks are made into anchors to enable an anchor successor.

It is also important that $\mathcal{R}$ not be too large. By making forks and joins anchors only when both subdags have principle anchors, it is possible to bound the number of anchor nodes. Proof of the following lemma follows the same ideas as bounding the number of internal nodes in a full binary tree with respect to the number of leaves. The "tree" here is nesting subdags of anchored forks.

LEMMA 3.4. *There are $O(|E_{non}|+1)$ anchor nodes (which are exactly the vertices in $\mathcal{R}$).*

Proof. Let p be the number of principle anchors. There are (at most) two principle anchors induced by each non-SP arc, so we have $p \leq 2|E_{non}|$. If $p = 0$, the only anchor is the first node in the graph. The rest of this lemma considers the case that $p \geq 1$.

To count the total number of anchors, we consider a collection of series-parallel graphs inductively over the series/parallel compositions used to create G_{SP}, starting from the base case singleton nodes. The idea is to count the number x of graphs containing principle anchors. Initially, $x = p$ as each principle anchor is in a separate graph. This number can only decrease when graphs are composed, since compositions do not create *principle* anchors.

The key observation is that whenever a composition rule results in more (fork or join) anchors, the number x of graphs with principle anchors decreases by one. In particular, anchors are only created on parallel composition of two graphs having principle anchors. In this case, 2 new anchor nodes are created. But two graphs with principle anchors are combined into one, so x decreases by 1. Thus, this case can occur at most $p-1$ times overall. No other cases results in anchor nodes being created, so the total number of anchor nodes including the source node is at most $p+2(p-1)+1 = 3p-1$.

Defining the anchor predecessor and successor. There is some flexibility in how to choose the anchor predecessor and successor of each node u in G_{SP}, as there may be multiple nodes that meet the requirements of Property 3.3. The algorithm chooses the nearest such node:

DEFINITION 1. *The anchor predecessor of v, denoted $PredAnchor(v)$, is the node y such that (1) $y \in V_{\mathcal{R}}$ is an anchor node, (2) $y \prec_{SP} v$ is a predecessor of v in G_{SP}, and (3) y is nearer, with respect to G_{SP}, than all other anchor nodes preceding v; that is, for all other anchor nodes $x \prec_{SP} v$, we have $x \prec_{SP} y$. Since the first node of the graph is an anchor node, the anchor predecessor is never null.*

The anchor successor is defined symmetrically except that it may be null if v has no successors that are anchor nodes.

It should be straightforward to see that, if the above definition is well-defined, i.e., if such a node always exists, then this definition of anchor predecessor and anchor successor satisfies Property 3.3. What may or may not be obvious is that the definition is well-defined.

LEMMA 3.5. *The definition of anchor predecessor (or successor) is well-defined. That is to say, for every node with any predecessors (or successors) that are anchors, there is exactly one such anchor node that is nearer than all others. Thus, the conditions of Property 3.3 are satisfied.*

Proof. Consider the anchor predecessor. (The argument for successor is similar.) Suppose for the sake of contradiction that there exists some node v for which the $PredAnchor(v)$ is ill-defined, i.e., there is no node meeting the requirements. Then there must be at least two distinct anchor nodes $x_1, x_2 \prec_{SP} v$ such that: there is no anchor node closer to v than x_1 or x_2. Consider any paths $p_1 = x_1 \overset{SP}{\rightsquigarrow} v$ and $p_2 = x_2 \overset{SP}{\rightsquigarrow} v$. Let u be the earliest node at which these paths cross (and possibly $u = v$). Then u must be a join node having anchor nodes in both of its subdags, so u would be an anchor node and $x_1, x_2 \prec_{SP} u$. This contradicts the assumption that there is no anchor node nearer to v than x_1 or x_2.

The edges in $\mathcal{R}$. The edges in $E_{\mathcal{R}}$ consist of all of the non-SP edges E_{non} plus just enough edges to ensure that the anchor nodes have the same transitive closure in both G and $\mathcal{R}$ (Property 3.1). Specifically, $\mathcal{R}$ consists of a series-parallel minor $\mathcal{R}_{SP} = (V_{\mathcal{R}}, E_{\mathcal{R}_{SP}})$ of G_{SP}, plus the non-SP edges E_{non}. Moreover, an anchor node is a fork or join in $\mathcal{R}_{SP}$ if and only if it is also a fork or join in G_{SP}.

For each anchor, its edges in $\mathcal{R}_{SP}$ are:

- For a non-join anchor node v, let (u,v) be the only incoming edge in G_{SP}. Then v has exactly one incoming edge in $\mathcal{R}_{SP}$: the edge $(PredAnchor(u), v)$.
- For a join anchor node j, let (ℓ, j) and (r, j) be the two incoming edges in G_{SP}. Then j has exactly two incoming edges in $\mathcal{R}_{SP}$: the edges $(PredAnchor(\ell), j)$ and $(PredAnchor(r), j)$.
- For a fork anchor node f, let (f, ℓ) and (f, r) be the two outgoing edges in G_{SP}. Then f has exactly two outgoing edges in $\mathcal{R}_{SP}$: the edges $(f, SuccAnchor(\ell))$ and $(f, SuccAnchor(r))$.

With the exception of the black-box reachability structure and making nodes into principle anchors, R-Sketch entirely ignores E_{non}, so we will generally reason about $\mathcal{R}_{SP}$.

We note that the only two black-box reachability structures are with respect to G_{SP} and $\mathcal{R}$; no query ever checks reachability on $\mathcal{R}_{SP}$ (after all, these would return the same answer as queries in G_{SP} but restricted to anchor nodes).

LEMMA 3.6. *This definition of $\mathcal{R}$ is well-defined. Moreover, it satisfies Property 3.1. That is, for any two anchor nodes $u, v \in V_{\mathcal{R}}$, we have $u \prec_{\mathcal{R}} v$ if and only if $u \prec_G v$.*

Proof. We start by showing that $\mathcal{R}$ is well defined. The only issue is with respect to the forks. We need to verify (1) that R-Sketch does not add any arcs of the form (f, NULL), and (2) that there are exactly two outgoing arcs. The latter is not immediately obvious given that arcs are also defined in the other direction. (1) follows from the fact that a fork is only an anchor if both subdags have anchors, and hence by Lemma 3.5 the sources ℓ and r of each subdag has an anchor successor. Moreover, for (2), $SuccAnchor(\ell)$ and $SuccAnchor(r)$ are "between" ℓ and r, respectively, and all other succeeding anchors. Thus, no other anchor would have an incoming arc originating at f.

We next show Property 3.1.

($\Rightarrow$) We will show the contrapositive. By definition, the anchor predecessor of a node precedes that node in G_{SP}. Similarly, the anchor successor is a successor of the node. By inspection of arcs added, $\mathcal{R}_{SP}$ only includes arcs between nodes that have the proper relationship in G_{SP}. That is to say, if $u \not\prec_{G_{SP}} v$ then $u \not\prec_{\mathcal{R}_{SP}} v$.

($\Leftarrow$) Suppose $u \prec_G v$. Then there exists a path p from u to v. Choose p to be the path that goes through the most possible anchors, and partition it at every anchor; that is, $p : u = x_0 \overset{G}{\rightsquigarrow} x_1 \overset{G}{\rightsquigarrow} \cdots \overset{G}{\rightsquigarrow} x_{k-1} \overset{G}{\rightsquigarrow} x_k = v$. We claim that for each subpath, $p_i : x_i \prec_{\mathcal{R}} x_{i+1}$, which is enough to imply $u \prec_{\mathcal{R}} v$. To prove the claim, suppose first that the $x_i \rightsquigarrow x_{i+1}$ contains a non-SP arc. The it consists of a single arc $x_i \xrightarrow{E_{non}} x_{i+1}$ because both endpoints of non-SP arcs are anchors. Otherwise, $x_i \overset{SP}{\rightsquigarrow} x_{i+1}$. By choice of p, x_i must be the anchor predecessor of the node that immediately precedes x_{i+1} on the path. (Or else there is a longer path.) Thus, the arc $(x_i, x_{i+1}) \in \mathcal{R}_{SP}$.

3.2 Traversally Incremental Construction Overview. This section outlines issues relating to constructing R-Sketch efficiently while executing the graph sequentially. The algorithm proceeds through the following steps: select a node u that has not yet executed; ***process*** u, by which we mean update the data structures; if all of u's predecessors have been executed, ***execute*** u, which corresponds to executing the original instructions in the program.[2] After executing u, the outgoing edges from u and corresponding nodes are revealed to the algorithm and may be selected. Initially, only the source node is available to select.

The main algorithm is in processing the nodes, which is where all the data-structural updates occur. In particular, we need to: keep track of the anchor predecessor; maintain the anchor successor; add nodes and edges G_{SP}, $\mathcal{R}$, and $\mathcal{R}_{SP}$ as appropriate; and update both reachability structures. The reachability structures are a black box implied by the underlying graphs G_{SP} and $\mathcal{R}$, so it suffices to specify when vertices and edges are added. The bulk of this section is devoted to discussing this processing step.

Note that the anchors, anchor predecessor, anchor successor, and $\mathcal{R}$, are all defined as in Section 3.1 but with respect to the subgraph of nodes processed thus far. R-Sketch maintains $\mathcal{R}_{SP}$ and $\mathcal{R}$ explicitly, but we shall describe only $\mathcal{R}_{SP}$—$\mathcal{R}$ is formed by adding the non-SP edges E_{non}.

As it turns out, maintaining the anchor predecessor is relatively easy, but keeping track of the anchor successor is more complicated. The algorithm does not keep the anchor successor directly, instead using a level of indirection. To achieve the performance bound, R-Sketch requires that the dag be traversed in a specific depth-first execution order, discussed next.

Execution Order. Where possible, R-Sketch selects nodes in the dag in depth-first, left-to-right order with respect to the series-parallel graph G_{SP}. That is to say, always process and execute as much of the left subdag of a fork node as possible before starting to process the right subdag of that fork. If there are no non-SP edges, this would mean executing the left subdag entirely before starting the right subdag, completing both before continuing to the join. However, because nodes are only ready to execute after their predecessors, it may be necessary to delay certain nodes due to dependencies on non-SP edges. We call such delays ***blocking***. A blocked node is executed whenever its dependencies are satisfied.

For concreteness, the execution operates as follows. Let S denote the depth-first, left-to-right sequential ordering of G_{SP} (i.e., ignoring any dependencies in E_{non}). On each step of the algorithm select the unexecuted, unblocked node u that is earliest in S.

For example, the nodes in Figure 1 are labeled according to their execution order used by R-Sketch. After executing node 12, we process the node 24. But node 24 cannot finish executing due to a dependency on 23. As such, 24 blocks and we continue with node 13 in depth-first order. After 23 is executed, 24 unblocks and can we resume the depth-first execution from there.

Challenges. The biggest challenge is that when a fork node is first encountered, it is impossible to predict if it will be

[2] Executing a node is where the race detector would perform queries into the data structure.

an anchor. A fork node only becomes an anchor when a principle anchor has been processed in both of the fork's subdags. Dealing with joins is easier because both subdags have executed completely before the corresponding join is processed, so we know immediately whether a join should be an anchor node.

Forks becoming anchors has a few ramifications. First, we must ensure that a fork node f is spliced into the right place in $\mathcal{R}_{SP}$. Second, arbitrarily many nodes may now have f as their anchor predecessor. Third, arbitrarily many nodes may now have f as their anchor successor. The worry is that all of these updates would be too expensive.

Consider, for example, the partial execution shown in Figure 2, just following node 12's execution. Next, node 24 is processed (but not executed because it blocks), so the structures should be updated as in Figure 3. Many changes have occurred due to the discovery of this new principle anchor. The fork node 7 becomes an anchor node, so it must be added to $\mathcal{R}_{SP}$, spliced in between 3 and 9. The newly discovered principle anchor 24 must also be added to $\mathcal{R}$, but this case is easier because it hangs off the end of the graph. In addition, many nodes have a new anchor predecessor or anchor successor. Specifically, the anchor predecessor of 7, 8, 10 and 12 changes from 3 to 7. The anchor successor of 2, 4, 5, 6, and 7 changes from 9 to 7. And the anchor successor of 12 changes from NULL to 24.

Anchor Predecessor/Successor and Proxies. R-Sketch stores the anchor predecessor explicitly and updates it through a graph search when new anchors are created. We shall argue that a node's anchor predecessor cannot change more than a constant number of times due to the depth-first execution order.

Maintaining anchor successors explicitly, however, would be too expensive. Consider again the running example. The anchor successors of 2, 4, and 5 change when 7 becomes an anchor. Looking at the *a posteriori* dag in Figure 1, we can see that these nodes will be updated again when 5 becomes an anchor. In fact, with a long chain of forks, the anchor successor may change a superconstant number of times.

Instead, R-Sketch maintains the anchor successors through a level of indirection. For each node u, R-Sketch stores a ***proxy*** node $proxy(u)$, defined as follows:

1. If u's anchor successor is null, then $proxy(u) = \text{NULL}$.
2. If u is an anchor node, then $proxy(u) = u$.
3. If u has an anchor successor, and there exists a join node j satisfying all of the following three properties, then $proxy(u) = PredAnchor(j)$. The properties are (1) u is in one of j's subdags, (2) that subdag does not contain any anchor nodes, and (3) the other subdag does contain at least one anchor node. (This is a case when j is not an anchor.)
4. Otherwise, $proxy(u) = PredAnchor(u)$.

In the example, $proxy(2) = proxy(4) = proxy(5) = proxy(6) = 3$ both before and after processing 24. Note that node 2 falls in the third case, whereas nodes 4, 5, and 6 fall in the fourth.

Given the proxy, anchor successor is computed as follows. The idea is to look at the proxy's outgoing edge in $\mathcal{R}_{SP}$. If there is more than one, i.e., the proxy is a fork, then look in the direction that would lead to the node.

```
GETSUCCANCHOR(u)
if u is an anchor node
    return u
elseif proxy(u) = NULL
    return NULL
else if proxy(u) is not a fork node
        return target of proxy(u)'s only out edge in R_SP
    elseif u is in proxy(u)'s left subdag in G_SP
                // i.e., check if proxy(u).left ≺_{G_SP} u
        return target of proxy(u)'s left out edge in R_SP
    else return target of proxy(u)'s right out edge in R_SP
```

In the example, 5's proxy does not change after processing 24, but the target of the proxy's outgoing edge *does* change in $\mathcal{R}$. This is exactly why the indirection of the proxy helps us — updates to $\mathcal{R}$ implicitly capture the changes to the anchor successor. For example, $proxy(5).out = 9$ before processing 24, and $proxy(5).out = 7$ after processing 24. We can see that for this example at least, GETSUCCANCHOR(5) returns the correct answer. The following lemma says that it is correct in general.

LEMMA 3.7. *Assuming $proxy(u)$ is maintained as defined,* GETSUCCANCHOR(u) *correctly returns $SuccAnchor(u)$.*

Proof. The cases that $proxy(u) = \text{NULL}$ or u is an anchor are trivial.

(Case 3.) Suppose the $proxy(u)$ is chosen according to the third (join-node) case. Then we claim that j and u have the same anchor successor. In particular, since u is in j's subdag, every path $u \overset{SP}{\leadsto} v$ to a successor v of j must pass through j. Thus, the only way u and j can have different anchor successors is if there is an anchor node on a path $u \overset{SP}{\leadsto} j$. That contradicts the assumption that we fall in this case.

Moreover, j must fall into the fourth case, i.e., $proxy(j) = PredAnchor(j) = proxy(u)$. The reason is that (i) j has an anchor successor, (ii) one of j's subdags contains an anchor, eliminating case 3 for j, and (iii) j is not an anchor join because its other subdag does not have an anchor. Thus, finishing case 3 reduces to applying case 4 on j.

(Case 4.) Suppose $proxy(u) = PredAnchor(u)$. Let x be the returned value from GETSUCCANCHOR. If $x =$

$SuccAnchor(u)$, we are done. Because $\mathcal{R}$ preserves reachability (Lemma 3.6), we must have $PredAnchor(u) \prec_{\mathcal{R}} x \prec_{\mathcal{R}} SuccAnchor(u)$. By definition of anchor predecessor and anchor successor and correctness of $\mathcal{R}$, the only options possible are $x = SuccAnchor(u)$ and $u \not\prec_{SP} x, x \not\prec_{SP} u$. If $x = SuccAnchor(u)$, we are done, so suppose instead that x and u are not related. Then the paths from $PredAnchor(u)$ to u and x must diverge at some point, i.e., there must be some fork $f \neq PredAnchor(u)$ and corresponding join $j \neq SuccAnchor(u)$ with $PredAnchor(u) \prec_{SP} f \prec_{SP} x \prec_{SP} j \prec_{SP} SuccAnchor(u)$ and $PredAnchor(u) \prec_{SP} f \prec_{SP} u \prec_{SP} j \prec_{SP} SuccAnchor(u)$. Since the fork and join are not anchors, u's branch must not have an anchor node. This is a contradiction as it would make u fall in Case 3.

3.3 Algorithm to Process Nodes. This section describes the algorithm for processing a node. Recall from Section 3.2 that the algorithm constructs $\mathcal{R}$ explicitly (adding nodes whenever new anchors are revealed), directly maintains $PredAnchor(u)$, and also maintains a value $proxy(u)$ that helps to determine the anchor successor. This section describes the changes to these structures when processing a node. We do not describe the reachability structures here — when we insert an edge or vertex into $\mathcal{R}_{SP}$ or $\mathcal{R}$, we implicitly mean to update the reachability structure on $\mathcal{R}$ as a black box. Moreover, the query algorithm was already described in Section 3.1, and the only remaining obstacle to correctness is ensuring that the algorithm correctly maintains what it is supposed to.

We break-up the description into multiple cases depending on the type of node being processed.

Case 1: Processing a node v that is neither a principle anchor nor a join. This is the easiest case. Let $(u,v) \in E_{SP}$ be the only incoming SP edge in G_{SP}. Set $PredAnchor(v) := PredAnchor(u)$ and $proxy(v) :=$ NULL. Note that this case captures v being a fork as well.

Case 2: Processing a join node j. Then there are two incoming edges $(\ell, j), (r, j) \in E_{SP}$, coming from the sinks of the j's left and right subdags in G_{SP}, respectively. Set $proxy(v) :=$ NULL. Next, we determine if j should become an anchor and set $PredAnchor(j) := p_j$, where p_j is computed as follows. Note that $u \prec_{SP} u$, so either of the first two cases covers equality:

(3.1)
$$p_j = \begin{cases} PredAnchor(r) & \text{if } PredAnchor(\ell) \prec_{SP} PredAnchor(r) \\ PredAnchor(\ell) & \text{if } PredAnchor(r) \prec_{SP} PredAnchor(\ell) \\ j & \text{otherwise} \end{cases}$$

If $p_j \neq j$, then we are done.

Otherwise ($p_j = j$), j is made into an anchor. Add the vertex j and the edges $(PredAnchor(\ell), j)$ and $(PredAnchor(r), j)$ to $\mathcal{R}_{SP}$. Update $proxy(j) := j$.

Finally, perform a backwards graph search in G_{SP} from ℓ and r, only visiting nodes x for which $proxy(x) =$ NULL, i.e., those predecessors with no anchor successor. Set $proxy(x) := PredAnchor(\ell)$ for those nodes encountered searching back from ℓ, and $proxy(x) := PredAnchor(r)$ for nodes encountered when searching back from r.[3]

Case 3: Processing a (non-join) node v that is incident on a non-SP edge. In the following, let (u,v) be v's only incoming edge in G_{SP}. This is the most complicated case because a fork can become an anchor node and anchor predecessors and successors for other nodes can change. We have two cases depending on whether a fork becomes an anchor. Following both cases, we deal with the impact of making v an anchor on other nodes.

Case 3a: $PredAnchor(u)$ has no outgoing edges in $\mathcal{R}_{SP}$. In our example, this case occurs, e.g., when processing node 9. (Figure 2 shows $\mathcal{R}$ after processing 9.)

In this case, no fork is made into an anchor. Simply add v and the edge $(PredAnchor(u), v)$ to $\mathcal{R}_{SP}$, and set $PredAnchor(v) := v$ and $proxy(v) := v$.

Case 3b: $PredAnchor(u)$ has an outgoing edge in $\mathcal{R}_{SP}$. In our example (see Figures 2 and 3), this case occurs when processing node 24.

First, identify the fork node by performing a backwards graph search from v in G_{SP} until reaching a node f with $proxy(f) \neq$ NULL.

Second, add f and v to $\mathcal{R}$ as follows. For the subsequent step, it will be convenient to reference certain old values, so temporarily store $p_f := PredAnchor(f)$ and $s_f :=$ GETSUCCANCHOR(f). Add f and v to $\mathcal{R}_{SP}$; replace the edge (f,v) by edges (p_f, f) and (f, s_f); add the edge (f,v).[4] The nodes f and v are now anchors, so set $PredAnchor(f) := f$, $proxy(f) := f$, $PredAnchor(v) := v$, and $proxy(v) := v$.

Next, update any nodes whose anchor predecessor and proxy should change. Specifically, perform a graph search forward from f in G_{SP}, truncating the search whenever reaching nodes x with $PredAnchor(x) \neq p_f$. Consider each node x reached during the search with $PredAnchor(x) = p_f$. For each of these nodes, update $PredAnchor(x) := f$. Also for each of these nodes, if $proxy(x) = p_f$, update it to $proxy(x) := f$.

In both cases: Add the non-SP edge to the reachability structure for $\mathcal{R}$. Perform a backwards graph search in G_{SP} from u, only visiting those nodes x for which $proxy(x) =$ NULL. Set $proxy(x) := PredAnchor(u)$ for those nodes.

[3]For all of these updated nodes, the anchor successor will never change again. We could explicitly store $SuccAnchor(x) := j$ in these cases. But since we need the proxy for other situations, we use it here as well.

[4]We do not have to remove the redundant edge (p_f, s_f) from the non-SP graph $\mathcal{R}$ as it does not change the transitive closure.

Correctness. The main correctness argument boils down to showing that the values maintained by the algorithm match the definitions. We have already shown that the defined structures are sufficient to support queries, so we get overall correctness as a corollary. The proof is tedious, with cases matching each case of the algorithm, but there is nothing surprising therein.

LEMMA 3.8. *The algorithm correctly maintains anchors, anchor predecessors, proxies, and the graph $\mathcal{R}_{SP}$ according to the definitions specified in Sections 3.1 and 3.2.*

Proof. By induction over processing nodes, with each case of the algorithm considered separately. Note that a non-anchor node has no impact on $\mathcal{R}$, the anchor predecessor, or the anchor successor of other nodes. Moreover, a node being processed has no successors. Thus, Case 1 trivially does what it needs to—update the info for the node v itself.

(*Case 2.*) The join j should only become an anchor if both subdags have anchors. If both subdags have an anchor, then those anchors cannot be related to each other. Thus, $p_j = j$ if and only if both subdags have anchors.

If $p_j \neq j$, then j correctly chooses its predecessor by inheriting the nearer anchor predecessor from its two incoming arcs. Nothing else changes.

If $p_j = j$, then j indeed becomes an anchor, and the arcs added to $\mathcal{R}_{SP}$ are exactly as defined. Since j has no successors, no other node's predecessor should change. But j may be the anchor successor of some nodes—specifically, only predecessors with no current anchor successors. If a node has an anchor successor, then so do all its predecessors, so the search can truncate at nodes with defined proxies. Consider just the search on the left subdag (the other being symmetric). For each node reached x, the backward path from j to x either lies along a path to $PredAnchor(\ell)$ or not. Note that no diverging path can have an anchor by definition of anchor predecessor. Thus, if x is on the path to the predecessor, x should have $proxy(x) = PredAnchor(x) = PredAnchor(\ell)$. If x is not on the path, then the path to x must follow a different branch from a join j', and x's branch cannot have any anchors, so we should have $proxy(x) = PredAnchor(j') = PredAnchor(\ell)$. Either way, the proxy is set correctly.

(*Claim: if a node has $proxy(x)$ set to a non-predecessor (i.e., the join case), then its proxy should never change again.*) Let j' be the join such that $proxy(x) = PredAnchor(j')$. All predecessors of j' have already executed, so x's situation with respect to j' cannot change.

The implication is that we only need to worry about updating the proxy again if it satisfies the other situations. Most notably, if $proxy(x) = PredAnchor(x)$, then $proxy(x)$ should change when $PredAnchor(x)$ changes.

(*Case 3.*) We first argue that case 3a and 3b correctly identify when a fork should become an anchor. If $PredAnchor(v)$ does not have any outgoing arcs in $\mathcal{R}_{SP}$, then there is no fork between $PredAnchor(v)$ and v with any anchor successor. If $PredAnchor(v)$ does have such an arc, then the path to the target of the arc must diverge at some fork f from the path to v. The backwards search finds the point of divergence: the nearest predecessor to v with an anchor successor. Thus, cases 3b correctly anchorizes fork nodes. In both cases 3a and 3b, the arcs added to $\mathcal{R}$ correspond exactly to the definition.

Adding f as an anchor only changes the anchor predecessor for those nodes that formerly had $PredAnchor(f)$ as their anchor predecessor. Since these must all be connected, the forward search corrects these. Moreover, according to the claim, the proxy should continue to track the anchor predecessor. No proxy or anchor predecessor for any node preceding f should change.

Making v itself an anchor is similar to (but slightly simpler than) case 2.

COROLLARY 3.1. *For any two already-processed nodes $u, v \in V$, the* QUERY(u,v) *correctly returns* TRUE *if $u \prec_G v$ and* FALSE *otherwise.*

Proof. By Lemma 3.3, we just need Properties 3.1 and 3.3. These are implied by Lemmas 3.5, 3.6 and 3.7 as long as the algorithm satisfies Lemma 3.8

3.4 Performance Analysis. This section argues that the construction of R-Sketch takes total time $O(n+k^2)$ when using the black-box routines from Lemmas 3.1 and 3.2, where $n = |V|$ is the number of vertices and $k = |E_{non}|$ is the number of non-SP edges.

The most worrisome part of the algorithm is that it performs graph searches which may, in the worst case, traverse the entire graph. Here we provide some lemmas that charge these searches against certain changes that can only occur a constant number of times.

The following lemma charges the cost of a backwards search against changing the node's proxy from NULL, which can only happen once. The key observation for Case 3 is that both backwards searches touch the same nodes.

LEMMA 3.9. *If backwards graph search (in Case 2 or Case 3) visits r nodes, then there are $\Omega(r)$ nodes whose proxies change from* NULL *to non-null.*

Proof. The backwards graph searches occur in Case 2 and Case 3. The former is easier as the search directly changes the proxy. The only nodes visited are those with $proxy(x) =$ NULL (and their neighbors to decide when to stop). Since the series-parallel graph has in-degree zero, this is $\Omega(1)$ nodes per change.

Case 3 performs up to two backwards search. The first does not directly change any proxies. However, it still only traverses nodes with a NULL proxy. These nodes will all be visited again in the second graph search, which matches the case. The cost here thus increases by a constant factor.

Bounding the cost of forward searches is less obvious. We state the two key helper lemmas here, followed by the main result: each node can only be involved in two forward searches, once when it is in the right subdag of a fork that becomes an anchor, and once when it is in the left subdag of a fork that becomes an anchor. An important assumption for these proofs is the depth-first execution order.

LEMMA 3.10. *Suppose the graph is processed in the specified depth-first order. Let f be a fork and let f_ℓ be a fork nested in f's left subdag. Suppose that both forks eventually become anchors. Then either the nested fork f_ℓ becomes an anchor before f does, or f becomes an anchor before processing the node f_ℓ.*

Proof. Suppose first we have not yet started processing f's right subdag at the point that f_ℓ is processed. Then by depth-first order, we cannot process any of f's right subdag until both of f_ℓ's subdags execute to the point they either complete (which means processing anchors by assumption) or block (also processing an anchor). So f_ℓ becomes an anchor before f_r.

Suppose instead that f's right subdag has started processing. Then at least one anchor must have already been processed in f's left subdag. The execution only jumps out of f's right subdag if a principle anchor is processed, either because that principle anchor is blocked or because some other node becomes unblocked. In either case, f has anchors in both subdags, so f becomes an anchor.

LEMMA 3.11. *Suppose the graph is processed in the specified depth-first order. Let f be a fork and let f_r be a fork nested in f's right subdag. Suppose that both forks eventually become anchors. Then f becomes an anchor before processing any of f_r's right subdag.*

Proof. The right subdag of f only starts executing when the left subdag has completed (having processed an anchor) or is blocked on anchors. In either case, by the time the first principle anchor is processed in f_r, f becomes an anchor. That would be while still processing f_r's left subtree.

LEMMA 3.12. *If processing nodes in the specified depth-first order, each node can be visited by at most 2 forward searches.*

Proof. Consider a particular node x. We claim that the two times x can be visited by forward searches are: (1) the first time that x is in the right subdag of a fork that becomes an anchor, and (2) the first time that x is in the left subdag of a fork that becomes an anchor.

By inspection, forwards searches only occur when forks become anchors. Moreover, there must be a new anchor in one of the fork's subdags to trigger the anchoring. Thus the corresponding join cannot have executed yet, and we need not worry about the forward search exiting the fork's subdags.

(Right subdag.) Suppose for the sake of contradiction that a particular node x is visited by two forward searches, from f and f_r, while in the right subdag of both forks. Without loss of generality, let f_r be in the right subdag of f. (Parallel composition has to nest.) Then by Lemma 3.11, f becomes an anchor before x is processed. This contradicts the assumption that x was involved in both searches.

(Left subdag.) Suppose for the sake of contradiction that a particular node x is visited by two forward searches, from f and f_ℓ, while in the left subdag of both forks. Without loss of generality, let f_ℓ be in the left subdag of f. Then by Lemma 3.10, we have two options. If f_ℓ becomes an anchor before f, then the forwards search from f would not pass through f, contradicting the assumption that both searches reach x. If not, f becomes an anchor before processing f_ℓ and hence before processing x, and the forward search from f cannot touch x.

We now give the main performance theorem. Substituting in the bound from Lemma 3.2, we get a total construction time of $O(n+k^2)$ and query time of $O(1)$.

THEOREM 3.1. *Let $n = |V|$ be the number of vertices and let $k = |E_{non}|$ be the number of non-SP edges in the input graph G.*

Consider any traversally incremental data structure that supports reachability queries on general graphs, supporting both edge insertions and queries. Let I be the total time to perform $\Theta(k)$ edge insertions, and let Q be the the time per query.

Then construction algorithm for R-Sketch runs in a total of $O(n+I)$ time, and R-Sketch answers reachability queries on any two already-processed nodes in $O(Q)$ time.

Proof. Almost all of the steps in construction algorithm to process each node is clearly constant time (e.g., looking at pointers, updating pointers, etc.). The exceptions are the following: (1) a constant number of insertions into R-Sketch for G_{SP}, (2) a constant number of insertions into R-Sketch for $\mathcal{R}$, (3) a constant number of reachability queries on G_{SP}, (4) the call to GETSUCCANCHOR, which is dominated by a reachability query on G_{SP}, and (5) the graph searches. Using an efficient data structure for the base SP graph (as in Lemma 3.1), all steps except the graph searches and the insertions into $\mathcal{R}$ require constant time per node processed.

The total number of arcs in $\mathcal{R}$ is a constant per node anchor node for the arcs in $\mathcal{R}_{SP}$, plus k for the non-SP arcs. By Lemma 3.4, the total number of arcs is thus $O(k)$, and hence the total cost of construction is $O(I)$.

To bound the backward searches in aggregate, we apply Lemma 3.9. Each only changes from $proxy = \text{NULL}$ once over the course of the construction, so we can charge the searches to those changes. The total cost of backward searches is thus $O(n)$.

We bound forward searches using Lemma 3.12, which says that each node can only be visited twice. Again we have a total cost of $O(n)$.

4 THE FULL RACE DETECTION ALGORITHM

The previous section discussed how to maintain the reachability data structure R-Sketch in a traversally incremental manner as the program executes. This section discusses how R-Sketch can be used to perform race detection in a parallel program with arbitrary synchronization and/or futures. In order to do so, we first review how a parallel program with particular programming constructs is modeled as a series parallel program with additional edges. We then discuss the other aspect of race-detection, namely the ***access history*** — for each memory location ℓ, the access history maintains enough information about the previous accesses to ℓ so that future accesses to ℓ can detect races.

Parallel programming constructs. As mentioned in Section 1 many parallel programming platforms support constructs for creating nested or fork-join parallelism. In this model, function F can ***spawn*** off a child function G, invoking the child without suspending the parent, thereby creating parallelism; similarly, F can invoke ***sync***, joining together all previously spawned children within the functional scope. The details of primitives differ, but at a high-level, the dependence structure generated is a series-parallel dag.

This paper considers on-the-fly race detection for a more general class of programs — programs that employ ***futures*** [4, 24]. Conceptually, the use of future involves two parallel primitives: ***create_future*** for creating parallelism, and ***get_future*** for joining parallel computation associated with the future. By preceding a function call to G in F with `create_future`, we create a ***future task***. The statement returns immediately with a ***future handle*** that promises to deliver the result of executing G without suspending F. Thus, F may continue to execute, possibly in parallel with G. In that regard, `create_future` and `spawn` share some similarities. Unlike `spawn`, however, parallel subcomputations (future tasks) created via `create_future` ***escape*** the scope of a `sync` — a subsequent `sync` joins together previously spawned functions but does not wait for future tasks (functions created by calls preceded by `create_future`) to return. Instead, one can invoke `get_future` on the future handle returned by `create_future` to join with the corresponding future task, and `get_future` ***blocks*** until the corresponding future task (G, in this case) finishes and a result is obtained. In short, the only guarantee is that the future task associated with a future handle h will finish executing before the invocation of `get_future` on h returns.

We now argue that our algorithm can be applied to programs that use futures — recall that our reachability data structure assumes that it gets a single series parallel dag with k extra arbitrary edges. It is pretty straightforward to see that a future task corresponds to a sub-SP-dag between `create_future` and `get_future` edges, whose other end is the node making the call. Therefore, a a program with that uses `spawn`, `sync`, `create_future` and `get_future` calls is essentially a set of series parallel dags connected to each other via edges corresponding to `create_future` and `get_future` calls. We can convert this to a single series parallel dag with extra edges by (a) adding an additional source and an additional sink node, and (b) artificially adding SP edges from each the this new source to the first node of each future task and from the last node of each future task to the new sink. This yields an SP-dag with extra non-SP edges for create-future and get-future calls. (The artificial edges are less restrictive than create/get edges, so adding them doesnt change the dags meaning. They are for modeling purposes only; the detector only needs to know a futures existence when create-future is called.) Now the root can have out-degree > 2, but this can be easily remedied. The total number of extra edges added is the total number of calls to `create_future` and `get_future`.

In order to perform race detection, simply execute the program sequentially in a left-to-right depth-first order — in particular, on a `spawn` or `create_future` of a function G, simply execute the function G eagerly while doing the appropriate book-keeping indicated by the algorithm. If `get_future` blocks, switch to the node that is next in the left-to-right depth first order. Note that eager execution guarantees that a `sync` never blocks. While executing the program, maintain another data structure, called the access history for every memory location — this is described next.

A different, but related paradigm, consists of using `put` and `await` constructs [9, 47] — a node that calls `await`(x) has a dependence from a node that calls `put`(x). It is easy to see that a programming model that allows these two calls in addition to `spawn` and `sync` allows series parallel dags with additional edges enforced due to `put` and `await`. Therefore, R-Sketch can also be used to detect races for these programs.

Access History. When performing race detection in a series parallel program, it is sufficient to store a constant number (1 for serial race detection, 2 for parallel race detection) of previous reader nodes and a single previous writer node in the access history [20, 36]. When a node s accesses memory, it checks if some subset (based on whether s is reading or writing) of these previous nodes are in parallel with s. Therefore, each memory access leads to at most a constant number of queries into the reachability data structure.

This property no longer holds for programs with futures, however. In particular, the access history for memory location ℓ still holds only one writer node, namely the most recent writer `last-writer`(ℓ). However, it must now store an arbitrarily large `reader-list`. Race detection proceeds as follows. Whenever a node s reads from the memory location ℓ, the race detector checks the reachability data structure to determine whether s is logically parallel with `last-writer`(ℓ); if so, a race is reported. Otherwise, s is added to `reader-list`(ℓ). When a node s writes to a memory location ℓ, the race detector must check s against all readers in `reader-list`(ℓ) and with `last-writer`(ℓ). If s is in parallel with any of them, then it declares a race. Otherwise, the reader list is set to and s is stored as `last-writer`(ℓ). Emptying the reader list in the absence of races does not miss future races, because anything that executes later that would be in parallel with these readers must also be in parallel with s (which is the new `last-writer`(ℓ)) and the race will be reported with s.

THEOREM 4.1. *The total running time of race detection for programs with k* `get_future` *calls is* $O(T_1 + k^2)$.

Proof. By Lemma 3.2 and Theorem 3.1, the total cost of maintaining the reachability data structure is $O(V + |V_{\mathcal{R}}| \cdot |E_{\mathcal{R}}|)$ where V is the total number of nodes in the computation, and V_R and E_R are both $O(k)$. In addition, the cost of each query is $O(1)$.

Queries are only performed at memory accesses. Each read requires one query — checking the reachability between the last writer and the current node. Each write may perform many queries — against all of the readers in the reader list. Note, however, that when a write occurs, all the readers in the `reader-list` are removed. Therefore, each read leads to at most two queries, once when the read itself occurs and once when a subsequent write to the same memory location occurs and the total number of queries are bounded by $2 \times$ number of reads. The total number of reads is at most T_1. Therefore, the total cost of race detection is $O(T_1 + k^2)$.

5 RELATED WORK

On-the-fly Race detection. As mentioned in Section 1, there is a large body of work on race detection for fork-join programs. Other structured computations have also been considered; Dimitrov et al. [18] propose an algorithm for race detection on computations that look like grids while Lee and Schardl [33] propose a race detector for fork-join computations that use a special kind of reduction mechanism. Recently, Surendran and Sarkar [45] proposed the first race detection algorithm for programs that use futures. Their reachability data structure has significantly more overhead than R-Sketch, however; in particular, the running time increases quadratically with the number of futures (that is multiplicatively instead of additively as for R-Sketch). There are two important distinctions between our approaches. First, their reachability data structure does not encode paths that include both SP and non-SP edges. Therefore, to answer a single reachability question of whether $u \prec v$, they must make multiple queries to the reachability data structure. Second, their reachability data structure explicitly stores a dag and each reachability query does a search on the dag; therefore, each query to the reachability data structure can take more than constant time.

In addition to race-detection for programs with structured parallelism and futures, there is a rich literature on dynamic race detection for programming models that generate computations with nondeterministic dependence structures, such as ones that involve locks [13, 14, 17, 19, 22, 39, 40, 43, 50, 52]. For such models, since the output necessarily depends on the schedule, the best correctness guarantee that a race detector can provide is for a given program, for a given input, and for a given *schedule*.

The Use of Futures. Blelloch et al. [7] propose to use futures to generate "non-linear pipelines," another form of parallelism that creates deterministic dependence structure and study scheduling bound for such programs. Their use of future falls under the structured use of futures. Others have looked at cache efficiency when one employs constructs that generate arbitrary dependencies such as unstructured use of futures [1, 44]. More recently, Herlihy and Liu [27] showed that, by restricting the use of futures, one can obtain a better cache efficiency than the unstructured use suggested by prior work. Their definition of structured futures is more restricted than ours — in particular, they enforce that future handles can only be passed as parameters to functions, but can not be returned as return values.

6 CONCLUSION

This paper provides a race-detection algorithm for futures that runs in $O(T_1 + k^2)$ time, with an additive overhead quadratic in the number of `get_future` operations. This algorithm uses a traversally incremental data structure, R-Sketch, for performing reachability queries. R-Sketch has a construction time of $O(n + k^2)$, where n is the number of nodes in the program dag and k is the number of future edges. Moreover, R-Sketch has a constant query time.

Note that if one is not careful, a program with futures can deadlock. Such a deadlock is deterministic, however, and does not depend on the schedule. In such cases, our algorithm race detects until the execution deadlocks.

Currently, R-Sketch requires that the dag be traversed in depth first order. Therefore, the race-detection algorithm must execute the computation serially. An interesting avenue of future work is how to parallelize race detection for programs with future. This would require a traversally incremental data structure that can support (1) a non-depth first insertion order; and (2) concurrent inserts and queries. Therefore, this extension appears to be non-trivial. Another avenue of future work is to implement the race detector and evaluate it in practice.

References

[1] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *Proceedings of the 12th ACM Annual Symposium on Parallel Algorithms and Architectures*, pages 1–12, 2000.

[2] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[3] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Frederico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, March 2009.

[4] Henry C. Baker, Jr. and Carl Hewitt. The incremental garbage collection of processes. *SIGPLAN Notices*, 12(8):55–59, 1977.

[5] Rajkishore Barik, Zoran Budimlić, Vincent Cavè, Sanjay Chatterjee, Yi Guo, David Peixotto, Raghavan Raman, Jun Shirako, Sağnak Taşırlar, Yonghong Yan, Yisheng Zhao, and Vivek Sarkar. The Habanero multicore software research project. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, pages 735–736, 2009.

[6] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *16th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 133–144, 2004.

[7] Guy E. Blelloch, Phillip B. Gibbons, Yossi Matias, and Girija J. Narlikar. Space-efficient scheduling of parallelism with synchronization variables. In *9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 12–23, 1997.

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

[9] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Sağnak Taşırlar. Concurrent collections. *Journal of Scientific Programming*, 18(3-4):203–217, August 2010.

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

[11] Rohit Chandra, Anoop Gupta, and John L. Hennessy. COOL: An object-based language for parallel programming. *IEEE Computer*, 27(8):13–26, August 1994.

[12] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538, 2005.

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

[14] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 258–269, 2002.

[15] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 937–946, 2002.

[16] John S. Danaher, I-Ting Angelina Lee, and Charles E. Leiserson. Programming with exceptions in JCilk. *Science of Computer Programming*, 63(2):147–171, December 2008.

[17] Joseph Devietti, Benjamin P. Wood, Karin Strauss, Luis Ceze, Dan Grossman, and Shaz Qadeer. RADISH: Always-on sound and complete race detection in software and hardware. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, pages 201–212, 2012.

[18] Dimitar Dimitrov, Martin Vechev, and Vivek Sarkar. Race detection in two dimensions. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 101–110, 2015.

[19] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, 2010.

[20] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. *Theory of Computing Systems*, 32(3):301–326, 1999.

[21] Jeremy T. Fineman. Provably good race detection that runs in parallel. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, MA, August 2005.

[22] Cormac Flanagan and Stephen N. Freund. FastTrack: Efficient and precise dynamic race detection. *SIGPLAN Not.*, 44(6):121–133, June 2009.

[23] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly threaded parallelism in manticore. *Journal of Functional Programming*, 20(5-6):537–576, November 2010.

[24] Daniel P. Friedman and David S. Wise. Aspects of applicative programming for parallel processing. *IEEE Transactions on Computers*, C-27(4):289–296, 1978.

[25] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 212–223, 1998.

[26] Robert H. Halstead, Jr. Multilisp: A language for concurrent

symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[27] Maurice Herlihy and Zhiyu Liu. Well-structured futures and cache locality. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 155–166, Orlando, Florida, USA, 2014.

[28] Intel® Cilk™ Plus. `https://www.cilkplus.org`, 2013.

[29] Intel Corporation. *Intel(R) Threading Building Blocks*, 2009. Available from `http://www.threadingbuildingblocks.org/documentation.php`.

[30] T. Kameda. On the vector representation of the reachability in planar directed graphs. *Information Processing Letters*, 3(3):75–77, 1975.

[31] David A. Kranz, Robert H. Halstead, Jr., and Eric Mohr. Mul-T: A high-performance parallel Lisp. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 81–90, 1989.

[32] I-Ting Angelina Lee, Silas Boyd-Wickizer, Zhiyi Huang, and Charles E. Leiserson. Using memory mapping to support cactus stacks in work-stealing runtime systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 411–420, 2010.

[33] I-Ting Angelina Lee and Tao B. Schardl. Efficiently detecting races in Cilk programs that use reducer hyperobjects. In *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 111–122, Portland, Oregon, USA, June 2015. ACM.

[34] Daan Leijen and Judd Hall. Optimize managed code for multi-core machines. *MSDN Magazine*, 2007. Available from `http://msdn.microsoft.com/magazine/`.

[35] Li Lu, Weixing Ji, and Michael L. Scott. Dynamic enforcement of determinism in a parallel scripting language. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 519–529, 2014.

[36] John Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pages 24–33, 1991.

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

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

[39] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 167–178, 2003.

[40] Eli Pozniansky and Assaf Schuster. MultiRace: Efficient on-the-fly data race detection in multithreaded C++ programs: Research articles. *Concurrency and Computation: Practice and Experience*, 19(3):327–340, March 2007.

[41] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Efficient data race detection for async-finish parallelism. In *Runtime Verification*, volume 6418 of *Lecture Notes in Computer Science*, pages 368–383. Springer Berlin / Heidelberg, 2010.

[42] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Scalable and precise dynamic datarace detection for structured parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 531–542, 2012.

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

[44] Daniel Spoonhower, Guy E. Blelloch, Phillip B. Gibbons, and Robert Harper. Beyond nested parallelism: Tight bounds on work-stealing overheads for parallel futures. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, pages 91–100, 2009.

[45] Rishi Surendran and Vivek Sarkar. Automatic parallelization of pure method calls via conditional future synthesis. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 20–38, 2016.

[46] Rishi Surendran and Vivek Sarkar. *Dynamic Determinacy Race Detection for Task Parallelism with Futures*, pages 368–385. Springer International Publishing, 2016.

[47] Sağnak Taşırlar and Vivek Sarkar. Data-driven tasks and their implementation. In *Proceedings of the 2011 International Conference on Parallel Processing*, pages 652–661, 2011.

[48] Robert Utterback, Kunal Agrawal, Jeremy Fineman, and I-Ting Angelina Lee. 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*, pages 83–94, 2016.

[49] Jacobo Valdes. *Parsing Flowcharts and Series-Parallel Graphs*. PhD thesis, Stanford University, December 1978. STAN-CS-78-682.

[50] Christoph von Praun and Thomas R. Gross. Object race detection. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 70–82, 2001.

[51] Haixun Wang, Hao He, Jun Yang, Philip S. Yu, and Jeffrey Xu Yu. Dual labeling: Answering graph reachability queries in constant time. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, pages 75–75, April 2006.

[52] Yuan Yu, Tom Rodeheffer, and Wei Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, pages 221–234, 2005.