On-the-fly Detection of Data Races for Programs with Nested Fork-Join Parallelism*

John Mellor-Crummey†
Center for Research on Parallel Computation
Rice University
Houston, TX 77251-1892
johnmc@rice.edu

Abstract

Detecting data races in shared-memory parallel programs is an important debugging problem. This paper presents a new protocol for run-time detection of data races in executions of shared-memory programs with nested fork-join parallelism and no other inter-thread synchronization. This protocol has significantly smaller worst-case run-time overhead than previous techniques. The worst-case space required by our protocol when monitoring an execution of a program P is $O(VN)$, where V is the number of shared variables in P, and N is the maximum dynamic nesting of parallel constructs in P's execution. The worst-case time required to perform any monitoring operation is $O(N)$. We formally prove that our new protocol always reports a non-empty subset of the data races in a monitored program execution and describe how this property leads to an effective debugging strategy.

1 Introduction

Parallel programs for shared-memory multiprocessors can exhibit *schedule-dependent* bugs, which cause erroneous behavior on some, but not all, execution schedules. The principal cause of such errors is unsafe or inadvertent communication through shared variables. If one thread of execution updates a shared variable concurrently with another thread's access to that variable, the program's behavior may depend on the temporal order of the accesses. Such concurrent accesses are known as "data races" or "access anomalies".

Pinpointing data races is difficult since adding diagnostic statements to a program can alter the relative timing of operations and change the set of execution schedules likely to occur. The act of trying to isolate a data race responsible for a schedule dependent error can cause the error to vanish. Thus, the technique used to debug sequential programs — re-executing them with instrumentation to provide information about program variable values — is likely to be ineffective for pinpointing data races in parallel program executions.

Three principal strategies have been proposed for isolating data races in parallel programs: static analysis, post-mortem analysis, and on-the-fly analysis.

Static analysis relies on classical dependence analysis of a program's text to determine when two references may refer to the same shared variable. Static techniques conservatively report dependences that include all potential data races that could occur during parallel execution. Strategies range from those that consider loop parallelism [1, 4], to those that consider more general tasking models [3, 14]. The conservative nature of static techniques, however, often leads to reports of data races that could never occur during execution. Experience with static analysis tools has shown that the number of false positives reported using these techniques is too high for programmers to rely exclusively on static methods for isolating data races. Combining static analysis with symbolic execution offers hope for reducing reports of infeasible races [15].

Post-mortem techniques for detecting data races involve collecting a log of events that occur during a program's execution and post-processing the log to look for evidence of data races [2, 5, 10]. If exhaustive logs are recorded, post-mortem techniques will report only feasible races. The primary drawback with post-mortem techniques is that execution logs can be enormous for parallel programs that execute for more than a trivial amount of time.

On-the-fly techniques involve augmenting a program to detect and report data races as they occur during its execution [6, 7, 9, 11, 12, 13]. These techniques maintain additional information at run-time to deter-

*This paper appeared in *Proc. of Supercomputing '91*, Albuquerque, NM, pages 24–33, November 1991.
†This work was supported in part by National Science Foundation Cooperative Agreement CCR-9045252.

mine when conflicting accesses to a shared variable have occurred. Like post-mortem techniques based on exhaustive logging, on-the-fly techniques report only feasible races. In general, on-the-fly techniques require less space than post-mortem techniques since much information can be discarded as an execution progresses.

On-the-fly techniques for detecting data races fall into two classes: summary methods [9, 12, 13] that report the presence of a data race with incomplete information about the references that caused it, and access history methods [7, 11] that can precisely identify each of a pair of accesses involved in a data race. From a programmer's standpoint, the precision of the information possible using access history methods is desirable for debugging. In the remainder of this paper, we focus on access history methods.

To pinpoint accesses involved in data races, access history methods maintain two types of information at run time: the threads (along with annotations identifying the source code statements involved) that have accessed each shared variable, and information that enables determination of whether any two threads are logically concurrent. When a thread t accesses a shared variable, the thread

1. determines if any thread in the history list performed an access that conflicts with t's current access,
2. reports a data race if a thread that made a conflicting access is logically concurrent with t,
3. removes from the history list the names of any threads that sequentially precede t in the execution and adds t to the list.

A drawback of previous access history protocols (*i.e.*, those used by Dinning & Schonberg's *Task Recycling* [6, 7] and Nudler & Rudolph's *English Hebrew Labeling* [11]) is that in the worst case, each shared variable's access history must contain names for as many as T threads — where T is the maximum amount of logical concurrency in the program — to guarantee that these protocols will never certify a program execution as *race free* when it actually contains a data race. The space requirements for maintaining such long access histories limit the usefulness of these techniques. In practice, approximations to these protocols have been implemented that maintain abbreviated access histories of length one or two [6, 7]; however, using abbreviated histories, these protocols can erroneously certify program executions as being free of data races.

In this paper we present a new access history protocol for detecting data races on the fly in executions of programs with nested fork-join parallelism. In contrast to previous access history protocols, our protocol bounds the length of each variable's history list by by a small constant that is program independent, yet our protocol ensures that if any data races exist in an execution, at least one will be reported. With this condition, an execution will never be erroneously certified as race free.

Bounding the length of history lists has two advantages. First, it reduces the worst-case space requirements. Second, it reduces the worst-case number of operations necessary to determine whether a thread's access is logically concurrent with any prior conflicting accesses.

Section 2 presents a graph model of fork-join program executions. This model serves as a framework for proving the correctness of our access history protocol. Section 3 presents *Offset-Span Labeling*, an on-line method for assigning names to threads in executions of programs with nested fork-join parallelism. Using Offset-Span Labeling, the concurrency relationship between any pair of threads can be inferred by comparing their names. Although similar to English-Hebrew Labeling [11], in the worst-case, Offset-Span Labeling assigns asymptotically shorter thread names, which lead to improved space and time bounds for access history protocols that use them. Section 4 presents our new protocol that uses bounded access histories to detect data races. Using properties of fork-join graphs and their respective Offset-Span labelings, we prove that if any data races exist in an execution of a program with nested fork-join parallelism but no other inter-thread synchronization, our protocol will report at least one data race for each shared variable involved in a race. Section 5 compares the time and space overhead of using our access history protocol and Offset-Span labels against the overhead with incurred using other access history methods. Section 6 describes the current status of this work and directions for future work.

2 A Model of Concurrency in Fork-Join Program Executions

This section defines fork-join graphs that model the run-time concurrency structure possible using closed, nestable fork-join constructs. Parallel Fortran programs that use nested parallel loops and sections are an instance of this programming model.

A *fork* operation terminates a thread and spawns a set of logically concurrent threads. Each *fork* operation has a corresponding *join* operation; when all of the threads descended from a fork terminate, the corresponding join succeeds and spawns a single thread. A thread participates in no synchronization operations other than the fork that spawned it and the join that terminates it. Each vertex in a fork-join graph represents a unique thread executing a (possibly empty) sequence of instructions. Each edge in a fork-join graph is induced by synchronization implied by a fork or join construct. A directed edge from vertex t_1 to vertex t_2 indicates that thread t_1 terminates execution before thread t_2 begins execution. Figure 1 shows a fragment of parallel Fortran and a fork-join graph that models the concurrency present during an execution of the code. Entering a parallel loop corresponds to a fork; exiting a parallel loop corresponds to a join. Each vertex in the fork-join graph is labeled with the sequence of code blocks whose execution it represents.

Before formally defining fork-join graphs, we define some useful notation for directed acyclic graphs (DAGs). In a DAG $G = (V, E)$, the *path relation* $x \rightsquigarrow_G y$ is true for $x, y \in V$ *iff* there is a path from x to y along edges in E; similarly $x \not\rightsquigarrow_G y$ is true *iff* there exists no directed path from x to y along edges in E. The *path star relation* $x \rightsquigarrow_G^* y$ is true for $x, y \in V$ *iff* $x \rightsquigarrow_G y \vee x = y$, namely there is a path from x to y along edges in E, or x and y are the same vertex.

Definition 1 constructively defines fork-join graphs which represent the concurrency relationships among threads in an execution of a fork-join program. Fork-join graphs are a subset of series-parallel graphs. The rules for constructing fork-join graphs ensure that no vertex has a singleton predecessor with outdegree 1. Such a pair of vertices would represent a pair of threads that execute sequentially. The rules for composing fork-join graphs collapse such pairs since their concurrency relationship is trivial.

Definition 1 *A* fork-join graph $G = (V, E, v_{src}, v_{snk})$ *is a DAG that*

- *has a designated source vertex* v_{src} *such that* $v_{src} \rightsquigarrow_G^* v$*, for all* $v \in V$.
- *has a designated sink vertex* v_{snk} *such that* $v \rightsquigarrow_G^* v_{snk}$*, for all* $v \in V$.
- *can be constructed using the following rules:*

1. *A singleton vertex* v *denotes a trivial fork-join graph* $G = (\{v\}, \emptyset, v, v)$.
2. *A compound fork-join graph can be formed in two ways:*

   parallel composition
   A set $S = \{G_i = (V_i, E_i, v_{src\,i}, v_{snk\,i}) | i = 1, n\}$ *of* $n \geq 2$ *disjoint fork-join graphs can be linked in parallel to form a new fork-join graph* $G = (V, E, v_{src}, v_{snk})$ *where*

$$\begin{array}{rcl} V & = & \{v_{src}, v_{snk}\} + \bigcup_{i=1,n} V_i \\ E & = & \bigcup_{i=1,n} \left(\begin{array}{l} E_i + \\ \{(v_{src}, v_{src\,i})\} + \\ \{(v_{snk\,i}, v_{snk})\} \end{array} \right) \end{array}$$

   series composition
   A pair of disjoint fork-join graphs, $G_1 = (V_1, E_1, v_{src1}, v_{snk1})$ *and* $G_2 = (V_2, E_2, v_{src2}, v_{snk2})$*, can be linked in series by merging vertices* v_{snk1} *and* v_{src2} *to form a new fork-join graph* $G = (V, E, v_{src1}, v_{snk2})$*, where*

$$\begin{array}{rcl} V & = & V_1 + V_2 - \{v_{src2}\} \\ E & = & E_1 + E_2 - \\ & & \{(v_{src2}, v) | (v_{src2}, v) \in E_2\} + \\ & & \{(v_{snk1}, v) | (v_{src2}, v) \in E_2\} \end{array}$$

The parallel composition rule describes how to link a set S of arbitrary fork-join graphs in parallel by nesting them inside a new, closed fork-join construct. The parallel composition rule adds two new threads v_{src}, the thread before a new fork, and v_{snk}, the thread after the corresponding new join, as well a synchronization edge from v_{src} to the source node of each fork-join graph in S, and a synchronization edge from the sink node of each fork-join graph in S to v_{snk}. In figure 1, the fork-join graph for each parallel loop is formed by parallel composition of the fork-join graph for each loop interation.

The series composition rule describes how to link a pair of arbitrary fork-join graphs in sequence by merging the sink vertex of the first graph with the source vertex of the second graph and retaining all of the edges. In figure 1, each node labeled "B,D" is the result of series composition of trivial fork-join graphs representing code blocks B and D respectively. Similarly, the fork-join graph that represents iteration `I=2` of the outermost parallel loop is the series composition of the fork-join graphs for the two loops nested inside.

Two vertices v_1 and v_2 in a fork-join graph G represent logically concurrent threads in an execution of a fork-join program *iff* $v_1 \not\rightsquigarrow_G^* v_2 \wedge v_2 \not\rightsquigarrow_G^* v_1$. The only ways this formula can be falsified is if v_1 and v_2 are not distinct, or if $v_1 \rightsquigarrow_G v_2 \vee v_2 \rightsquigarrow_G v_1$. If the vertices are not distinct, the threads are the same and thus not concurrent. In the second case, the vertices are related by a path of directed edges. The interpretation of a directed edge (as described earlier) as temporal precedence and the transitivity of this precedence relation for paths of edges means that v_1 and v_2 could not in fact be concurrent if they are connected by a path of directed edges.

```
[code block A]
PARALLEL DO I=2,4
  [code block B]
  IF (I.EQ.2) THEN
    PARALLEL DO J = 1,2
      [code block C]
    ENDDO
  ENDIF
  [code block D]
  PARALLEL DO J=1,I
    [code block E]
  ENDDO
  [code block F]
ENDDO
[code block G]
```

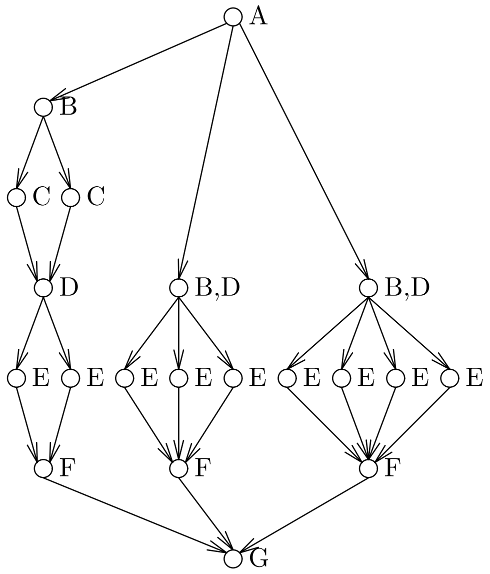

Figure 1: A fragment of parallel Fortran and its corresponding fork-join graph.

To facilitate inductive proofs about fork-join graphs, we define $rule(G)$ to be the minimum number of applications of the series and parallel composition rules needed to construct a fork-join graph G from a set of trivial fork-join graphs. (It is important to define $rule(G)$ to be the minimum number of rule applications since applying series composition to a pair of trivial fork-join graphs results in another trivial fork-join graph.)

3 Offset-Span Labeling

Offset-Span labeling is an on-line scheme for labeling each thread in a fork-join program execution. Each thread's label contains information that identifies its position in a corresponding fork-join graph. By comparing the labels of two threads, their concurrency relationship can be deduced. Offset-Span labeling is similar to Nudler and Rudolph's *English-Hebrew labeling* [11]. In both Offset-Span (OS) and English-Hebrew (EH) labeling, a thread in an execution of a fork-join program computes its own unique label using only local information — specifically, the label(s) of its immediate predecessor(s) in a fork-join graph. (In contrast, the Task Recycling technique [6, 7] requires a centralized data structure to maintain information about free task descriptors. It is preferable to avoid use of centralized data structures in parallel programs since they tend to introduce serial bottlenecks.) In both EH and OS labeling, the length of a thread's label increases along with the nesting depth of fork-join constructs. Also, both schemes use a lexicographic-style comparison of labels to determine if the threads they represent are concurrent.

An advantage of OS labeling is that its definition guarantees that the length of a thread's OS label is *always* proportional to the current nesting depth of the fork-join pair surrounding the thread. The length of the OS label for a thread following a join is always equal to the length of the OS label for the thread that executed the matching fork. Using EH labeling (as described in [7]), the length of a thread's label can grow in proportion to the number of fork operations encountered along the execution path leading to the creation of the thread; the length of an EH label following a join is greater than the length of the EH label for the thread that executed the matching fork. Dinning and Schonberg mention the existence of a heuristic [7, p. 4] that reportedly limits the length of EH labels to the level of nesting. It is important to minimize the length of labels used by these methods since shorter labels reduce the space required to store them at execution time as well as the time spent comparing them.

Definition 2 *An* Offset-Span labeling *of a fork-join graph G assigns a label consisting of a non-null sequence of ordered pairs to each of the vertices of G. Each ordered pair $[o, s]$ consists of two components: the* offset *and the* span*. The span indicates the number of threads spawned by an s-way fork from which this label pair is descended. The offset distinguishes among relatives descended from the same parent. An OS labeling of a fork-join graph $G = (V, E, v_{src}, v_{snk})$ is computed* *as follows given an initial OS label for v_{src} that consists of a non-null sequence of offset-span pairs:*

1. *For a vertex $v \in V$ of outdegree $n > 1$ (v is the source node of some fork-join subgraph of G) that has an OS label of L, where L is some non-empty sequence of label pairs (hereafter, we use the notation $OSL(v) = L$): let v_i denote the ith child of v, $0 \leq i < n$ (the ordering of the children is insignificant). Assign $OSL(v_i) = L[i, n]$, where juxtaposition of L and $[i, n]$ implies concatenation.*

2. *For a vertex v of indegree $n > 1$ (v is the sink node of some fork-join subgraph of G) that has some labeled vertex v' as a predecessor: $\exists_{L,u,w,x,y} OSL(v') = L[u, w][x, y]$, where L is a (possibly null) sequence of label pairs. (In a fork-join graph, $OSL(v')$ must be of this form. Any node in G other than v_{src} or v_{snk} has an OS label consisting of at least two label pairs. By the definition of fork-join graphs, v_{src} cannot be the predecessor of any vertex of indegree > 1 and v_{snk} cannot be the predecessor of any vertex.) Assign $OSL(v) = L[u + w, w]$.*[1]

The labeling is complete since by the composition rules no vertex in a fork-join graph can have a predecessor of outdegree 1 and itself be of indegree 1. The labeling is consistent since the composition rules guarantee that any vertex that is a successor of a vertex with outdegree > 1 has indegree 1. Comparisons between two labels are made by comparing the corresponding ordered pairs in the label sequences from left to right. Each thread's OS label in an execution of a program with closed, nestable fork-join parallelism can be computed on line efficiently from the label of its predecessor. Figure 2 shows an OS labeling of the fork-join graph shown in figure 1.

The following lemma shows the relationship between the labels assigned to the source and sink of a fork-join graph. Note that this lemma also implies that the length of each thread's OS label assigned using the rules of definition 2 is directly proportional to the nesting depth of fork-join constructs surrounding the thread.

Lemma 1 *In an OS labeling of a fork-join graph $G = (V, E, v_{src}, v_{snk})$, if v_{src} has a label $P[o, s]$, where P is an arbitrary (possibly null) sequence of ordered label pairs and o and s are arbitrary constants, then v_{snk} has label $P[o', s]$, for some o' such that $o \bmod s = o' \bmod s$.*

[1] This label is the same regardless of the predecessor v' chosen. The label of a sink node for a fork-join subgraph is determined by the label of the corresponding source node. (See Lemma 1.)

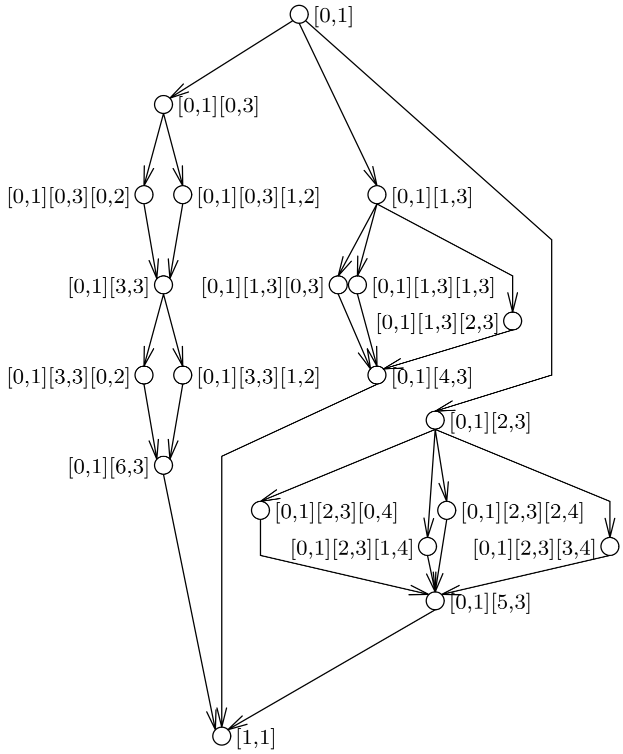

Figure 2: An Offset-Span Labeling of a fork-join graph.

Proof Induction on the size of G as measured using $rule(G)$.

Base Case. For any trivial graph G ($rule(G) = 0$), the lemma is satisfied with $o' = o$.

Induction Hypothesis. Assume that the lemma holds for every fork-join graph G with $rule(G) < k$.

Induction Step. Show that the lemma holds for each fork-join graph G with $rule(G) = k$. We consider applying each of the composition rules to a collection of $G_i, i = 1, n$ ($n \geq 2$) fork-join graphs with $\sum_{i=1}^{n} rule(G_i) = k - 1$.

series An application of the series composition rule to form $G = (V, E, v_{src1}, v_{snk2})$ from 2 disjoint fork-join graphs $G_1 = (V_1, E_1, v_{src1}, v_{snk1})$ and $G_2 = (V_2, E_2, v_{src2}, v_{snk2})$, where $rule(G_1) + rule(G_2) = k-1$. By the induction hypothesis, the lemma holds for both G_1 and G_2 separately. Let v_{src1} have OS label $P[o, s]$. By the induction hypothesis, v_{snk1} has label $P[o_1, s]$, where $o \bmod s = o_1 \bmod s$. Let v_{src2} have OS label $P[o_1, s]$, by the induction hypothesis v_{snk2} has label $P[o_2, s]$, where $o_1 \bmod s = o_2 \bmod s$. The series composition rule merges v_{snk1} with v_{src2}. After the merge, the labeling remains consistent. Since v_{src2} has no incoming edges in G_2, the label of the merged node is completely determined by the labels of the ancestors of v_{snk1} in G_1; thus, the label of the merged node remains $P[o_1, s]$. Since v_{snk1} has no outgoing edges in G_1, the outdegree of the merged node in G remains the same as that of v_{src2} in G_2. Therefore, the labels of the descendants of v_{src2} remain the same. By transitivity, $o \bmod s = o_2 \bmod s$ and the lemma is satisfied for graph G.

parallel An application of the parallel composition rule to form $G = (V, E, v_{src}, v_{snk})$ from a set $S = \{G_i = (V_i, E_i, v_{srci}, v_{snki}) | i = 1, n\}$ of $n \geq 2$ disjoint fork-join graphs, where $\sum_{i=1}^{n} rule(G_i) = k-1$. By the induction hypothesis, the lemma holds for both each G_i separately. Let v_{srci} have OS label $P'[i-1, n]$. By the induction hypothesis v_{snki} has label $P'[o_i, n]$, where $(i-1) \bmod n = o_i \bmod n$. The parallel composition rule links v_{src} to each $v_{srci},\ i = 1, n$ and links $v_{snki},\ i = 1, n$ to v_{snk}. Let the OS label of v_{src} be $P[o, s]$. Letting $P' = P[o, s]$ makes the all of the labels of nodes in subgraphs $G_i,\ i = 1, n$ consistent with the labeling rules. By labeling rule 2, v_{snk} is assigned label $P[o+s, s]$ since its ancestors $v_{snki},\ i = 1, n$ have OS labels $P'[o_i, n] = P[o, s][o_i, n]$ respectively. The lemma is satisfied for G since $o \bmod s = (o+s) \bmod s$.

The lemma follows by the principle of induction. □

In the following lemma, we show that by comparing the OS labels for a pair of threads in an execution, it is straight-forward to determine if one thread has finished before a second thread begins (*i.e.*, the vertices representing the threads are related by $\rightsquigarrow_G$ in the fork-join graph G representing the execution).

Lemma 2 *Given the OS labeling of a fork-join graph* $G = (V, E, v_{src}, v_{snk})$, $x \rightsquigarrow_G y$, *is true for* $x, y \in V$ iff *one of the following properties holds for their respective OS labels, OSL(x) and OSL(y)*

case 1 $\exists_{P,S}(OSL(x) = P) \wedge (OSL(y) = PS)$ *where both P and S are any non-null sequence of ordered label pairs.*

case 2
$\exists_{P,S_x,S_y,o_x,o_y,s}(OSL(x) = P[o_x, s]S_x) \wedge (OSL(y) = P[o_y, s]S_y) \wedge (o_x < o_y) \wedge (o_x \bmod s = o_y \bmod s)$ *where P, S_x, and S_y are (possibly null) sequences of ordered pairs.*

Proof Any fork-join graph that contains more than one vertex must have been constructed through some sequence of applications of the parallel and series composition rules. Let $G_s = (V_s, E_s, v_{srcs}, v_{snks})$ be the smallest fork-join subgraph of G that contains both x and y. Case 1 holds iff G_s was constructed from a set of disjoint fork-join graphs using the parallel composition rule, $x = v_{srcs}$, and $y \in V_s - \{v_{srcs}, v_{snks}\}$. Case 2 holds iff (a) G_s was constructed from a set of disjoint fork-join graphs using parallel composition, $x \in V_s - \{v_{snks}\}$, and $y = v_{snks}$, or (b) G_s was constructed by linking some pair of disjoint fork-join graphs using series composition. In case 2, P is *null iff* $G_s = G$, S_x is *null iff* $x = v_{srcs}$, and S_y is *null iff* $y = v_{snks}$. The enumeration of ancestor relationships covered by these cases is complete. Case 1 and 2a cover all ancestor relationships if the last rule applied to form G_s was the parallel composition rule. In these cases x has to be v_{srcs} or y has to be v_{snks}, otherwise G_s would not be the smallest subgraph that contains both x and y with $x \rightsquigarrow_G y$. Case 2b covers all possible ancestor relationships if the last rule applied to form G_s was the series composition rule. □

Below, we define a *left of* relation that defines a partial ordering of vertices in a fork-join graph that are not related by the $\rightsquigarrow_G^*$ relation (*i.e.*, vertices that represent concurrent threads). The access history protocol described in section 4 requires a labeling scheme for which a left-of relation can be defined. English-Hebrew labels contain sufficient information to compute a left-of relation, but labels assigned by the Task Recycling technique do not. Here we define a left-of relation for OS labels.

Definition 3 *For an OS labeling of a fork-join graph* $G = (V, E, v_{src}, v_{snk})$, *the* "left of" relation, *denoted* $x \prec_G y$, *is true for* $x, y \in V$ iff *the following property holds for their OS labels OSL(x) and OSL(y)*

$\exists_{P,S_x,S_y}(OSL(x) = P[o_x, s]S_x) \wedge (OSL(y) = P[o_y, s]S_y) \wedge (o_x \bmod s < o_y \bmod s)$, *$P$ is a non-null sequence of ordered label pairs, S_x and S_y are (possibly null) sequences of ordered label pairs.*

The left-of relation establishes a canonical ordering of relatives with respect to their lowest common ancestor.

4 A Protocol for Detecting Data Races

Two accesses to the same variable are *conflicting* if at least one of them is a write. A data race in the execution of a fork-join program exists when two or more concurrent threads perform conflicting accesses to the same shared variable. In terms of the fork-join graph model, a data race exists in an execution if two threads represented by vertices v_i and v_j in a fork-join graph G perform conflicting accesses to the same shared variable and $v_i \not\rightsquigarrow_G^* v_j \wedge v_j \not\rightsquigarrow_G^* v_i$ (the threads are unordered by synchronization, and thus their executions are logically concurrent).

To detect data races on the fly, each potentially unsafe access to a shared variable during a parallel program execution must be monitored. A program transformer must allocate access history storage for each shared variable with a reference that is the endpoint of a dependence carried by a parallel construct (*i.e.*, static analysis was unable to prove that some reference by a logically concurrent thread does not result in a conflicting access to the variable). At each variable reference that is an endpoint of a dependence carried by a parallel construct, the transformer must add a call to a monitoring protocol that inspects and updates the variable's access history. The transformer must also insert statements that enable each thread to compute a label that reflects its concurrency relationship to other threads. At execution time, the monitoring protocol reports any logically concurrent, conflicting accesses to a shared variable.

For an execution of a fork-join program, the existence of a data race involving a shared variable is solely a function of which threads access it and the concurrency relationship between the threads that is implied by the fork and join constructs in the program. Therefore, we can consider data races for each shared variable independently.

We define an *access interleaving* to model a set of accesses to a shared variable by threads in a fork-join program.

Definition 4 *An* access interleaving *for a shared variable X by threads whose run-time concurrency relationship is modeled by a fork-join graph $G = (V, E, v_{src}, v_{snk})$ is denoted I_G^X. I_G^X consists of a totally ordered sequence of accesses $A_1, \ldots, A_n$. Each access is performed by some thread; let $v_G(A) \in V$ be the vertex in G that represents the thread that performed the access A. An access $A_i \in I_G^X$ marks vertex $v_G(A_i)$ with either an X_{read} or an X_{write} token. Multiple accesses in I_G^X may mark the same vertex, and a vertex can be marked with both X_{read} and X_{write} tokens. No access $A_j \in I_G^X$ may mark a vertex $v_1 \in V$ if some $A_i \in I_G^X$, $i < j$ previously marked a vertex $v_2 \in V$ such that $v_1 \rightsquigarrow_G v_2$.*

The definition of an access interleaving assumes *sequentially consistent* [8] shared memory. We refer to an access in I_G^X as a *read* if it marks a vertex with an X_{read} token, or as a *write* if it marks a vertex with an X_{write} token.

In the remainder of this section, we present protocols for detecting data races caused by conflicting accesses to a single shared variable and prove their correctness. We formulate the problem of on-the-fly detection of data races as detecting conflicting, logically concurrent accesses in an access interleaving for a shared variable. An access interleaving I_G^X for a variable X and a fork-join graph $G = (V, E, v_{src}, v_{snk})$ is

checkread(`access_history`, `thread_label`)
  if `access_history^.W`$_{\texttt{last}}$ $\not\rightsquigarrow_G^*$ `thread_label` **then**
    report a WRITE-READ **data race**
  endif
  if `thread_label` $\prec_G$ `access_history^.R`$_{\texttt{ll}}$ **or**
      `access_history^.R`$_{\texttt{ll}}$ $\rightsquigarrow_G$ `thread_label` **then**
    `access_history^.R`$_{\texttt{ll}}$:= `thread_label`
  endif
  if `access_history^.R`$_{\texttt{lr}}$ $\prec_G$ `thread_label` **or**
      `access_history^.R`$_{\texttt{lr}}$ $\rightsquigarrow_G$ `thread_label` **then**
    `access_history^.R`$_{\texttt{lr}}$:= `thread_label`
  endif
end checkread

Figure 3: Monitoring protocol for a read.

checkwrite(`access_history`, `thread_label`)
  if `access_history^.W`$_{\texttt{last}}$ $\not\rightsquigarrow_G^*$ `thread_label` **then**
    report a WRITE-WRITE **data race**
  endif
  if `access_history^.R`$_{\texttt{ll}}$ $\not\rightsquigarrow_G^*$ `thread_label` **or**
      `access_history^.R`$_{\texttt{lr}}$ $\not\rightsquigarrow_G^*$ `thread_label` **then**
    report a READ-WRITE **data race**
  endif
  `access_history^.W`$_{\texttt{last}}$:= `thread_label`
end checkwrite

Figure 4: Monitoring protocol for a write.

checked if for each access $A \in I_G^X$

- if A is a read the **checkread** protocol (figure 3) is called with a pointer to X's access history and the label for thread $v_G(A)$ (the thread performing the access), and
- if A is a write the **checkwrite** protocol (figure 4) is called with a pointer to X's access history and the thread label for $v_G(A)$.

The **checkread** and **checkwrite** protocols determine whether an access by the current thread is involved in a data race with any access earlier in the interleaving. Any thread labeling scheme is suitable for use with this protocol as long as the $\rightsquigarrow_G$, $\rightsquigarrow_G^*$, and $\prec_G$ relations can be determined using label comparisions.

If **checkread** is invoked when any thread reads a shared variable X, the protocol guarantees that the $\texttt{R}_{\texttt{lr}}$ component of X's access history contains the label for the "lowest", "rightmost" thread, and $\texttt{R}_{\texttt{ll}}$, the label for the "lowest", "leftmost" thread. The concepts of "lowest", "rightmost", and "leftmost" are well-defined for threads in an execution modeled by a fork-join graph G in terms of the $\rightsquigarrow_G$, $\rightsquigarrow^*_G$, and $\prec_G$ relations. If **checkwrite** is invoked when any thread writes to X, the protocol guarantees that the $\mathtt{W_{last}}$ component of X's access history contains the label for the thread that last performed a write to that variable.

The theorems that follow in this section show that the **checkread** and **checkwrite** protocols guarantee that if an access interleaving contains one or more data races, at least one of these races will be detected and reported. Thus, using these protocols, an execution will be reported free of races *iff* no data races are present.

Theorem 1 *In a checked access interleaving I^X_G for a variable X and a fork-join graph $G = (V, E, v_{src}, v_{snk})$,* **checkwrite** *will report a data race for a write in I^X_G if it is logically concurrent with some earlier read in I^X_G.*

Proof Suppose $r \in I^X_G$ marks $R \in V$ with an X_{read} token, $w \in I^X_G$ marks $W \in V$ with an X_{write} token, r precedes w in I^X_G, and R and W are logically concurrent, but **checkwrite** fails to report a data race for w.

Without loss of generality, assume that vertices in V are named by their thread labels. If **checkwrite** reports no race for w, then it must be the case that $\mathtt{R_{lr}} \rightsquigarrow^*_G W \wedge \mathtt{R_{ll}} \rightsquigarrow^*_G W$ when **checkwrite** is called for w (*i.e.*, W is not logically concurrent with previous readers $\mathtt{R_{lr}}$ or $\mathtt{R_{ll}}$ saved by the **checkread** protocol).

Since **checkread** has been executed for each *read* preceding w in the interleaving (including r), we are guaranteed that

$$\begin{aligned} &\mathtt{R_{ll}} \preceq_G R \wedge \mathtt{R_{ll}} \not\rightsquigarrow_G R \wedge \\ &R \preceq_G \mathtt{R_{lr}} \wedge \mathtt{R_{lr}} \not\rightsquigarrow_G R \wedge \\ &\mathtt{R_{ll}} \not\rightsquigarrow_G \mathtt{R_{lr}} \wedge \mathtt{R_{lr}} \not\rightsquigarrow_G \mathtt{R_{ll}} \end{aligned} \quad (1)$$

It must be the case that $R \not\rightsquigarrow^*_G \mathtt{R_{lr}} \wedge R \not\rightsquigarrow^*_G \mathtt{R_{ll}}$; otherwise, by transitivity of the $\rightsquigarrow^*_G$ relation, $R \rightsquigarrow^*_G W$, which violates the supposition that R and W are logically concurrent. This implies $R \neq \mathtt{R_{ll}} \wedge R \neq \mathtt{R_{lr}}$. Using this to refine (1) we can conclude that if such an R exists,

$$\mathtt{R_{ll}} \prec_G R \wedge R \prec_G \mathtt{R_{lr}} \quad (2)$$

If $\mathtt{R_{lr}} = \mathtt{R_{ll}}$, then (2) is not satisfiable and there can be no R concurrent with W; therefore, if such an R exists

$$\mathtt{R_{lr}} \neq \mathtt{R_{ll}} \quad (3)$$

Let $G_s = (V_s, E_s, v_{srcs}, v_{snks})$ be the smallest fork-join subgraph of G that contains both $\mathtt{R_{ll}}$ and $\mathtt{R_{lr}}$. By (1) and (3), $\mathtt{R_{ll}} \not\rightsquigarrow^*_G \mathtt{R_{lr}} \wedge \mathtt{R_{lr}} \not\rightsquigarrow^*_G \mathtt{R_{ll}}$; therefore, $v_{srcs} \neq \mathtt{R_{ll}} \wedge v_{srcs} \neq \mathtt{R_{lr}}$. A corollary of this is that $|V_s| > 1$ which implies $rule(G_s) > 0$. The composition rule last applied to construct G_s could not have been the series composition rule. The condition that G_s is the smallest fork-join graph containing both $\mathtt{R_{ll}}$ and $\mathtt{R_{lr}}$ would imply that one vertex must be in each of the components linked in series; this contradicts (1) since $\mathtt{R_{ll}}$ and $\mathtt{R_{lr}}$ would be related by $\rightsquigarrow_G$. Therefore, G_s must have been formed from some set S of disjoint fork-join graphs using the parallel composition rule. Both $\mathtt{R_{ll}}$ and $\mathtt{R_{lr}}$ cannot belong to the same element of S, otherwise G_s would not be the smallest fork-join graph containing them both. Therefore, v_{srcs} is the closest common ancestor of $\mathtt{R_{ll}}$ and $\mathtt{R_{lr}}$, and v_{snks} is their closest common descendant. Since $\mathtt{R_{lr}} \rightsquigarrow_G W$ and $\mathtt{R_{ll}} \rightsquigarrow_G W$, then $v_{snks} \rightsquigarrow^*_G W$. As justified below, v_{srcs} must be an ancestor of R (*i.e.*, $v_{srcs} \rightsquigarrow_G R$):

- If $R \rightsquigarrow^*_G v_{srcs}$, then $R \rightsquigarrow_G \mathtt{R_{ll}} \wedge R \rightsquigarrow_G \mathtt{R_{lr}}$. By transitivity of the path relation, $R \rightsquigarrow_G W$, contradicting the supposition that R and W are logically concurrent.
- If R is to the left of v_{srcs}, then by definition of $\prec_G$, $R \prec_G \mathtt{R_{ll}}$, contradicting (1).
- If v_{srcs} is to the left of R, then by definition of $\prec_G$, $\mathtt{R_{lr}} \prec_G R$, contradicting (1).

Also, $v_{snks} \not\rightsquigarrow_G R$, otherwise, by transitivity $\mathtt{R_{lr}} \rightsquigarrow_G R \wedge \mathtt{R_{ll}} \rightsquigarrow_G R$, contradicting (1). By the definition of closed, nestable fork-join graphs, every descendant of a source vertex that is not a descendant of the corresponding sink vertex must be an ancestor of the sink vertex. Therefore, since $v_{srcs} \rightsquigarrow_G R \wedge v_{snks} \not\rightsquigarrow_G R$, then $R \rightsquigarrow_G v_{snks}$. But then by transitivity, $R \rightsquigarrow_G W$, contradicting the supposition that R and W are logically concurrent.

By showing a contradiction in every case to the supposition that there can exist some read r that precedes a write w in I^X_G such that they mark logically concurrent vertices but **checkwrite** fails to report a data race for w, the theorem is proven. □

Theorem 2 *In a checked access interleaving I^X_G for a variable X and a fork-join graph $G = (V, E, v_{src}, v_{snk})$, if any two writes in I^X_G are logically concurrent,then* **checkwrite** *will report a data race.*

Proof Suppose vertices $V_w \subseteq V$ are marked with X_{write} tokens by accesses in I^X_G and at least one pair of vertices in V_w is concurrent. Two writes in an access interleaving are *adjacent* if there is no other write between them in the sequence. If the vertices marked by each pair of adjacent writes in I^X_G are related by the path star relation $\rightsquigarrow^*_G$, by transitivity of $\rightsquigarrow^*_G$ no pair of writes in V_w would be concurrent. By the original supposition, at least two of the vertices in V_w are concurrent; therefore, some pair of vertices $v_1, v_2 \in V_w$ that are marked by a pair of adjacent writes in I_G^X must not be related by $\rightsquigarrow_G^*$. Without loss of generality, let v_1 be the vertex marked by the first of the adjacent writes; thus, $v_1 \not\rightsquigarrow_G^* v_2$. Since the writes by v_1 and v_2 are adjacent, `Wlast` will contain the thread label for v_1 when **checkwrite** is called for the following write by v_2; **checkwrite** will report a data race since $v_1 \not\rightsquigarrow_G^* v_2$. We have shown that if write accesses in I_G^X mark any two concurrent vertices in G, then a data race will be reported, thus proving the theorem. □

Theorem 3 *In a checked access interleaving I_G^X for a variable X and a fork-join graph $G = (V, E, v_{src}, v_{snk})$, a data race will be reported if a read in I_G^X is logically concurrent with some earlier write in I_G^X.*

Proof Suppose $w \in I_G^X$ marks $W \in V$ with an X_{write} token, $r \in I_G^X$ marks $R \in V$ with an X_{read} token, W precedes R in I_G^X, and W and R are logically concurrent, but no data race is reported.

Without loss of generality, assume that vertices in V are named by their thread labels. If there is no intervening write between w and r in I_G^X, when **checkread** executes for r, `Wlast` $= W$ and **checkread** will report a data race since by supposition W and R are concurrent.

If there is some sequence of writes $w_1, \ldots, w_n$ between w and r in I_G^X then it cannot be the case that $W \rightsquigarrow_G^* v_G(w_1)$, $v_G(w_i) \rightsquigarrow_G^* v_G(w_{i+1})$ for $1 \leq i < n$, and $v_G(w_n) \rightsquigarrow_G^* R$; otherwise by transitivity of the $\rightsquigarrow_G^*$ relation $W \rightsquigarrow_G^* R$, contradicting our original supposition that they are concurrent. If $W \rightsquigarrow_G^* v_G(w_n)$, then $v_G(w_n) \not\rightsquigarrow_G^* R$, otherwise W and R could not be concurrent. In this case, at vertex R, `Wlast` would contain the label for $v_G(w_n)$ and **checkread** would report a data race between $v_G(w_n)$ and R. Otherwise, if $W \not\rightsquigarrow_G^* v_G(w_n)$, then w is concurrent with w_n and by theorem 2 **checkwrite** will report at least one data race for some pair of adjacent writes in the subsequence of I_G^X beginning with w and ending with w_n. □

Theorem 4 *In a checked access interleaving I_G^X for a variable X and a fork-join graph $G = (V, E, v_{src}, v_{snk})$, at least one data race will be reported if there are any conflicting, logically concurrent accesses in I_G^X.*

Proof There are three cases of conflicting accesses to consider,

1. a read is concurrent with a write, and the read precedes the write in I_G^X,
2. two writes are concurrent,
3. a read is concurrent with a write, and the write precedes the read in I_G^X.

By theorem 1, a data race will be reported for any concurrent accesses in case 1. By theorem 2 a data race will be reported for any concurrent accesses in case 2. Finally, by theorem 3, a data race will be reported for any concurrent accesses in case 3. □

Theorem 4 shows that if any data races are present in an access interleaving for a shared variable, at least one will be reported using our **checkread** and **checkwrite** access history protocols. By applying the solution to detect any races for an individual shared variable to each of the shared variables in a program, we can guarantee that if a program execution exhibits any data races given a particular input, then the **checkread** and **checkwrite** protocols will report at least one data race for each shared variable that is actually involved in a race during that execution.

Using the monitoring protocol described in this section leads to an effective debugging strategy for eliminating data races from a program execution for a given input. Run the program on the given input with the monitoring protocol in place. Each time a data race is reported (the access history protocol precisely reports both endpoints of the race), fix the cause of the data race, and re-execute the program with the same input. Since the access history procotols given in this section will report data races (if any exist) regardless of the interleaving order, the protocol can be used to check for races in a program that is executed in a canonical serial order. Executing programs in a canonical serial order while debugging is often convenient as it provides the user with deterministic behavior that simplifies the task of determining the origin of variable values that indirectly caused a data race to occur.

If no race is detected in an execution, then no race will occur in *any* execution of the program for that particular input and the program is guaranteed to be deterministic *for that input.* The key insight behind this observation is that the only thing that could cause an execution for the given input to behave differently would be if there were some form of non-determinism present. Data races are the sole source of non-determinism in programs that have nested fork-join parallelism but no other inter-thread synchronization. Therefore, if no data race is detected in one execution of such a program for a given input, then no data race can exist in any execution for that input.

Practical implementations of the **checkread** and **checkwrite** protocols described in this section must respect the underlying assumptions upon which the correctness proofs are based. In particular, all updates and inspections of an access history by the **checkread** and **checkwrite** protocols must be coordinated. Without coordinating updates to a variable's access history, the **checkread** protocol could not correctly maintain the invariants with respect to $\mathtt{R_{1r}}$ and $\mathtt{R_{1l}}$. The simplest coordination strategy is enforcing mutually exclusive access. Such coordination could cause bottlenecks if there is pervasive read sharing of a variable among concurrent threads. By using dependence analysis to limit monitoring instrumentation to only the cases in which read-write conflicts seem imminent, hopefully such bottlenecks could be avoided. Other less restrictive coordination strategies appear possible, but it would be necessary to relax some of the invariants maintained by the protocols and show that data races are guaranteed to be detected even with relaxed invariants.

5 Analysis

In this section we examine the space and time complexity of using our access history protocol with Offset-Span labels and compare it to the complexity of the protocols described in the literature for English-Hebrew Labeling [11] and Task Recycling [6, 7]. To be consistent with the notation of Dinning and Schonberg [6], we present our analysis in terms of the following parameters:

T	—	maximum logical concurrency
V	—	number of monitored shared variables
N	—	maximum level of fork-join nesting
B	—	total number of threads in an execution

Table 1 compares the worst case time and space complexity of the earlier access history methods, English-Hebrew Labeling and Task Recycling, with the worst case time and space complexity of our access history protocol using Offset-Span labels.

For the EH Labeling and Task Recycling access history protocols described in the literature, each monitored variable has an access history that may contain as many as T thread names if the variable is accessed by each thread that is active when the program attains its maximum logical concurrency; this leads to the VT term in the their space complexities. The second term in the space complexity of Task Recycling arises because each thread has an associated "parent vector" of length T that is used to summarize the concurrency relationships between a thread and its ancestors. Since T threads may be active simultaneously, T^2 space may be needed. In EH Labeling, the size of an EH label for a thread is proportional to the nesting depth of fork-join constructs which is bounded by N. (This analysis assumes the existence of an effective heuristic alluded to by Dinning and Schonberg [7, p. 4] that limits the length of labels to $O(N)$. Without the heuristic, labels can grow arbitrarily long. A description of the heuristic was unavailable to the author of this paper at the time of this publication.) If access histories store pointers to EH labels, each label is at most of length N, and there can be at most VT distinct pointers to labels. If reference counting garbage collection is used, the maximum space used to store EH labels is bounded by $O(VTN)$. If the number of threads in a program execution B is less than VT, then this places a tighter bound on the space to store the labels of $O(BN)$ since at most one label per thread needs to be stored.

In the expression for the worst-case space complexity for our new access history protocol using Offset-Span labels, the first term accounts for the constant size access history for each monitored variable. The second term reflects the space needed to store OS labels. If access histories store pointers to OS labels, each label is at most of length N, and there can be at most $O(V)$ distinct pointers to OS labels. If reference counting garbage collection is used, the maximum space used to store OS labels is bounded by $O(VN)$. If the number of threads in a program execution B is less than V, then this places a tighter bound on the space to store the labels of $O(BN)$ since at most one label per thread needs to be stored.

The worst case time to verify whether an individual access to a variable is involved in a data race is $O(TN)$ for the EH Labeling protocol since an access may need to be compared against T entries in the variable's access history and each comparison may take $O(N)$ time. For Task Recycling, the worst case time to verify whether an individual access to a variable is involved in a data race is $O(T)$; the parent vector representation in Task Recycling enables access comparisons in constant time, but a comparison may be needed for each of T entries in a variable's access history. For our new access history protocol with Offset-Span labels, the corresponding time is only $O(N)$ since the label for the current access need only be compared with a constant number of other labels.

The worst-case time overhead at thread creation for EH and OS labeling is $O(N)$ for assignment of a label of size $O(N)$ to a thread. Task Recycling incurs worst-case overhead of $O(T)$ at thread creation and termination since a parent vector of size $O(T)$ may need to be created for a new thread, and when theads meet at a join, their parent vectors of size $O(T)$ must be merged.

Since T is typically greater than 2^N, using our new access protocol represents a significant worst-case savings in both space and time over earlier protocols for on-the-fly detection of data races.

Algorithm	Space	Time	
		Thread Creation & Termination	Per Access
Task Recycling	$O(VT + T^2)$	$O(T)$	$O(T)$
EH Labeling	$O(VT + \min(BN, VTN))$	$O(N)$	$O(NT)$
OS Labeling	$O(V + \min(BN, VN))$	$O(N)$	$O(N)$

Table 1: Comparison of Worst Case Time and Space Requirements.

6 Status and Future Work

A prototype system for dependence-based instrumentation of potential data races in parallel FORTRAN programs has been developed as part of the debugging system in the ParaScope Programming Environment [4]. The instrumentation system inserts calls to a run-time library that uses Offset-Span Labeling and the access history protocol described in section 4. The prototype instrumentation system currently handles simple programs with loop based parallelism. Currently, procedure calls from within parallel loops are not handled. Ongoing implementation efforts are focused on extending interprocedural analysis in ParaScope so that the dependence-based instrumentation can interprocedurally propagate requirements for instrumentation into procedures called from within parallel constructs. Once the interprocedural instrumentation system is complete, the on-the-fly debugging system will be useful for more than toy programs.

Future work includes extending the access history protocol and proofs to handle regular patterns of synchronization such as sections in DOACROSS loops and the PCF FORTRAN generalization of this construct: ordered sequence synchronization. Preliminary indications are that the protocols will extend naturally to accommodate this larger class of programs.

Acknowledgments

I thank the referees for the improvements they suggested and I am indepted to the referee who pointed out several erratta. Robert Hood and Seema Hiranandani participated in early discussions of these ideas. Robert Hood implemented the prototype dependence-based instrumentation system.

References

[1] R. Allen, D. Baumgartner, K. Kennedy, and A. Porterfield. PTOOL: A semi-automatic parallel programming assistant. In *Proc. of the 1986 International Conference on Parallel Processing*, pages 164–170, Aug. 1986.

[2] T. R. Allen and D. A. Padua. Debugging fortran on a shared memory machine. In *Proc. of the 1987 International Conference on Parallel Processing*, pages 721–727, Aug. 1987.

[3] W. F. Appelbe and C. E. McDowell. Anomaly reporting – a tool for debugging and developing parallel numerical applications. In *Proc. First International Conference on Supercomputers*, FL, Dec. 1985.

[4] V. Balasundaram, K. Kennedy, U. Kremer, K. McKinley, and J. Sublok. The ParaScope editor: An interactive parallel programming tool. In *Proc. Supercomputing '89*, pages 540–550, Reno, NV, Nov. 1989.

[5] J.-D. Choi, B. P. Miller, and R. H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, 1991.

[6] A. Dinning and E. Schonberg. An evaluation of monitoring algorithms for access anomaly detection. Ultracomputer Note 163, Courant Institute, New York University, July 1989.

[7] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 1–10, Mar. 1990.

[8] L. Lamport. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9), Sept. 1979.

[9] S. L. Min and J.-D. Choi. An efficient cache-based access anomaly detection scheme. In *Proc. of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 235–244, Palo Alto, CA, Apr. 1991.

[10] R. H. B. Netzer and B. P. Miller. Detecting data races in parallel program executions. In D. Gelernter, T. Gross, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*. MIT Press, 1991. Also in *Proc. of the 3rd Workshop on Prog. Langs. and Compilers for Parallel Computing*, Irvine, CA, (Aug. 1990).

[11] I. Nudler and L. Rudolph. Tools for efficient development of efficient parallel programs. In *First Israeli Conference on Computer Systems Engineering*, 1988. Cited in [7].

[12] E. Schonberg. On-the-fly detection of access anomalies. In *Proc. ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 285–297, June 1989.

[13] G. L. Steele, Jr. Making asynchronous parallelism safe for the world. In *Proc. of the 1990 Symposium on the Principles of Programming Languages*, pages 218–231, Jan. 1990.

[14] R. N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26(5):362–376, May 1983.

[15] M. Young and R. N. Taylor. Combining static concurrency analysis with symbolic execution. *IEEE Transactions on Software Engineering*, 14(10):1499–1511, Oct. 1988.