Scalable and Precise Dynamic DataRace Detection for Structured Parallelism

Raghavan Raman  Jisheng Zhao  Vivek Sarkar
Rice University  Rice University  Rice University
raghav@rice.edu  jisheng.zhao@rice.edu  vsarkar@rice.edu

Martin Vechev  Eran Yahav *
ETH Zürich  Technion
martin.vechev@inf.ethz.ch  yahave@cs.technion.ac.il

Abstract
Existing dynamic race detectors suffer from at least one of the following three limitations:
(i) space overhead per memory location grows linearly with the number of parallel threads [13], severely limiting the parallelism that the algorithm can handle.
(ii) sequentialization: the parallel program must be processed in a sequential order, usually depth-first [12, 24]. This prevents the analysis from scaling with available hardware parallelism, inherently limiting its performance.
(iii) inefficiency: even though race detectors with good theoretical complexity exist, they do not admit efficient implementations and are unsuitable for practical use [4, 18].

We present a new precise dynamic race detector that leverages structured parallelism in order to address these limitations. Our algorithm requires constant space per memory location, works in parallel, and is efficient in practice. We implemented and evaluated our algorithm on a set of 15 benchmarks. Our experimental results indicate an average (geometric mean) slowdown of $2.78 \times$ on a 16-core SMP system.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Software/Program Verification—reliability, validation; D.2.5 [Software Engineering]: Testing and Debugging—monitors, testing tools; D.3.4 [Programming Languages]: Processors—debuggers; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—program analysis

General Terms Algorithms, Languages, Verification

Keywords Parallelism, Program Analysis, Data Races

1. Introduction
Data races are a major source of errors in parallel programs. Complicating matters, data races may occur only in few of the possible schedules of a parallel program, thereby making them extremely hard to detect and reproduce. The importance of detecting races has motivated significant work in the area. We briefly summarize existing race detectors and the main contributions of our approach below.

Existing Race Detectors  FastTrack is a state-of-the-art parallel race detection algorithm which handles classic unstructured fork-join programs with locks [13]. While versatile, a key drawback of FastTrack is its worst-case space overhead of $O(n)$ per instrumented memory location, where $n$ is the number of threads in the program. This space overhead implies that the algorithm can typically only be used with a small number of parallel threads. Increasing the number of threads can quickly cause space overheads and slowdowns that render the algorithm impractical. FastTrack applies some optimizations to reduce the overhead, but even for locations that are read shared, the algorithm maintains $O(n)$ space. Unfortunately, in domains where structured parallelism dominates, programs typically use a massive number of lightweight tasks (e.g., consider a parallel-for loop on a GPU) and often the parallel tasks share read-only data.

There have been various proposals for race detectors targeting structured parallel languages, notably SP-bags [12] and All-Sets [8] for Cilk and its extension ESP-bags [24] for subsets of X10 [7] and Habanero-Java (HJ) [6]. The SP-bags, All-Sets, and ESP-bags algorithms only need $O(1)$ space per instrumented memory location but are limited in that they must always process the parallel program in a depth-first sequential manner. This means that the algorithms cannot utilize and scale with the available hardware parallelism. The SP-hybrid algorithm for Cilk [4] is an attempt to address the sequentialization limitation of the SP-bags algorithm. However, despite its good theoretical bounds, the SP-hybrid algorithm is very complex and incurs significant inefficiencies in practice. The original paper on SP-hybrid [4] provides no evaluation and subsequent evaluation of an incomplete implementation of SP-hybrid [18] was done only for a small number of processors; a complete empirical study for SP-hybrid has never been done. However, the inefficiency is clear from the fact that the CilkScreen race detector used in Intel Cilk++ [1] has chosen to use the sequential All-Sets algorithm over the parallel but inefficient SP-hybrid. Further, the SP-hybrid algo-
Structured Parallelism

Structured parallel programming simplifies the task of writing correct and efficient parallel programs in two ways. First, a wide range of parallel programs can be succinctly expressed with a few well-chosen and powerful structured parallel constructs. Second, the structure of the parallel program can be exploited to provide better performance, for instance, via better scheduling algorithms. Third, structured parallelism often provides guarantees of deadlock-freedom. Examples of languages and frameworks with structured parallelism include Cilk [5], X10 [7], and Habanero Java (HJ) [6].

Our Approach

A key idea is to leverage the structured parallelism to efficiently determine whether conflicting memory accesses can execute in parallel. Towards that end, we present a new data structure called the Dynamic Program Structure Tree (DPST). With our algorithm, the time overhead for every monitoring operation is independent of the number of tasks and worker threads executing the program. Similarly to FastTrack, SP-bags and ESP-bags, our algorithm is sound and precise for a given input: if the algorithm does not report a race for a given execution, it means that no execution with the same input can trigger a race (i.e. there are no false negatives). Conversely, if a race is reported, then the race really exists (i.e. there are no false positives). These properties are particularly attractive when testing parallel programs as it implies that for a given input, we can study an arbitrary program schedule to reason about races that may occur in other schedules. As we will demonstrate later, our algorithm is efficient in practice and significantly outperforms existing algorithms.

Main Contributions

The main contributions of this paper are:

- A dynamic data race detection algorithm for structured parallelism with the following properties:
  - works in parallel,
  - uses only constant space per monitored memory location.
- A data structure called the Dynamic Program Structure Tree (DPST) that keeps track of relationships between tasks and can be accessed and modified concurrently.
- An efficient implementation of the algorithm together with a set of static optimizations used to reduce the overhead of the implementation.
- An evaluation on a suite of 15 benchmarks indicating an average (geometric mean) slowdown of 2.78× on a 16-core SMP system.

Rest of the paper is organized as follows: Section 2 discusses the structured parallel setting. Section 3 presents the dynamic program structure tree (DPST). Section 4 introduces our new race detection algorithm, Section 5 presents the details of the implementation of our algorithm and the optimizations that we used to reduce the overhead. Section 6 discusses our experimental results. Section 7 discusses related work and Section 8 concludes the paper.
We assume standard operational semantics of async/finish constructs as defined in FX10 [20]. The semantics of statements and expressions other than async/finish is standard [30]. That is, each transition represents either a basic statement, an expression evaluation or the execution of an async or a finish statement. For our purposes, given a trace, we assume that the execution of each statement is uniquely identified (if a statement executes multiple times, each dynamic instance is uniquely identified). We refer to an execution of a statement as a dynamic statement instance. We say that a statement instance is an async instance if the statement performs an async operation. Similarly for finish instances.

**Definition 1** (Step). A step is a maximal sequence of statement instances such that no statement instance in the sequence includes an async or a finish operation.

**Definition 2** (DPST). The Dynamic Program Structure Tree (DPST) for a given execution is a tree in which all leaves are steps, and all interior nodes are async and finish instances. The parent relation is defined as follows:

- Async instance A is the parent of all async, finish, and step instances directly executed within A.
- Finish instance F is the parent of all async, finish, and step instances directly executed within F.

There is a left-to-right ordering of all DPST siblings that reflects the left-to-right sequencing of computations belonging to their common parent task. Further, the tree has a single root that corresponds to the implicit top-level finish construct in the main program.

### 3.1 Building a DPST

Next we discuss how to build the DPST during program execution. When the main task begins, the DPST will contain a root finish node F and a step node S that is the child of F. F corresponds to the implicit finish enclosing the body of the main function in the program and S represents the starting computation in the main task.

**Task creation** When a task T performs an async operation and creates a new task T_{child}:

1. An async node A_{child} is created for task T_{child}. If the immediately enclosing finish (IEF) F of T_{child} exists within task T, then A_{child} is added as the rightmost child of F. Otherwise, A_{child} is added as the rightmost child node of (the async) node corresponding to task T.
2. A step node representing the starting computations in task T_{child} is added as the child of A_{child}.
3. A step node representing the computations that follow task T_{child} in task T is added as the right sibling of A_{child}.

Note that there is no explicit node in a DPST for the main task because everything done by the main task will be within the implicit finish in the main function of the program and hence all of the corresponding nodes in a DPST will be under the root finish node.

**Start Finish** When a task T starts a finish instance F:

1. A finish node F_n is created for F. If the immediately enclosing finish F' of F exists within task T (with corresponding finish node F'_n in the DPST), then F_n is added as the rightmost child of F'_n. Otherwise, F_n is added as the rightmost child of the (async) node corresponding to task T.
2. A step node representing the starting computations in F is added as the child of F_n.

**End Finish** When a task T ends a finish instance F, a step node representing the computations that follow F in task T is added as the right sibling of the node that represents F in the DPST.

Note that the DPST operations described thus far only take O(1) time. Thus, the DPST for a given program run grows monotonically as program execution progresses and new async, finish, and step instances are added to the DPST. Note that since all data accesses occur in steps, it follows that all tests for whether two accesses may happen in parallel will only take place between two leaves in a DPST.
Figure 2. A part of a DPST. LCA is the Lowest Common Ancestor of steps S1 and S2. A is the DPST ancestor of S1 which is the child of LCA. S1 and S2 can execute in parallel if and only if A is an async node.

Example We can now return to the example program in Figure 1 and study its steps and final DPST. Note the way statement instances are grouped into steps. When the main task starts executing finish F1, a node corresponding to F1 is added as the root node of the DPST, and a step node step1 is added as the child of F1; step1 represents the starting computations in F1, i.e., instances of statements S1 and S2. When the main task forks the task A1, an async node corresponding to A1 is added as the right-most child of F1 (since the immediately enclosing finish of F1 is F1 and it is within the main task), a step node step2 is added as the child of A1, and a step node step5 is added as the right sibling of A1. step2 represents the starting computations in A1 (i.e., instance of statements S3, S4, and S5) and step5 represents the computation that follows A1 in the main task (i.e., instances of statements S9, S10, and S11). After this point, the main task and the task A1 can execute in parallel. Eventually, the DPST grows to the form shown in the figure.

3.2 Properties of a DPST

In this section, we briefly summarize some key properties of a DPST. The proofs of these properties have been omitted due to space limitations, but can be found in [25].

- For a given input that leads to a data-race-free execution of a given async-finish parallel program, all executions of that program with the same input will result in the same DPST.
- Let F be the DPST root (finish) node. Each non-root node n0 is uniquely identified by a finite path from n0 to F:
  \[ n_0 \xrightarrow{r_{i_0}} n_1 \xrightarrow{r_{i_1}} n_2 \xrightarrow{r_{i_2}} \ldots \xrightarrow{r_{i_{k-1}}} n_k \]
  where \( k \geq 1 \), \( n_k = F \), and for each \( 0 \leq i < k \), \( n_i \) is the \( r_i \)th child of node \( n_{i+1} \). The path from \( n_0 \) to \( F \) stays invariant as the tree grows. For a given statement instance, its path to the root is unique regardless of which execution is explored (as long as the executions start with the same state). This property holds up to the point that a data race (if any) is detected.
- The DPST is amenable to efficient implementations in which nodes can be added to the DPST in parallel without any synchronization in O(1) time. One such implementation is described in Section 5.

Definition 3. A node A is said to be to the left of a node B in a DPST if A appears before B in the depth first traversal of the tree.

As mentioned above, even though the DPST changes during program execution, the path from a node to the root does not change and the left-to-right ordering of siblings does not change. Hence, even though the depth first traversal of the DPST is not fully specified during program execution, the left relation between any two nodes in the current DPST is well-defined.

Definition 4. Two steps, S1 and S2, in a DPST \( \Gamma \) that corresponds to a program \( P \) with input \( \psi \), may execute in parallel if and only if there exists at least one schedule \( \delta \) of \( P \) with input \( \psi \) in which \( S_1 \) executes in parallel with \( S_2 \).

The predicate \( DMHP(S_1, S_2) \) evaluates to true if steps \( S_1 \) and \( S_2 \) can execute in parallel in at least one schedule of a program and to false otherwise (DMHP stands for “Dynamic May Happen in Parallel” to distinguish it from the MHP relation used by static analyses). We now state a key theorem that will be important in enabling our approach to data race detection.

Theorem 1. Consider two leaf nodes (steps) \( S_1 \) and \( S_2 \) in a DPST, where \( S_1 \not\approx S_2 \) and \( S_1 \) is to the left of \( S_2 \) as shown in Figure 2. Let LCA be the node denoting the least common ancestor of \( S_1 \) and \( S_2 \). Let node A be the ancestor of \( S_1 \) that is the child of LCA. Then, \( S_1 \) and \( S_2 \) can execute in parallel if and only if A is an async node.

Proof. Please refer to [25].

Example Let us now look at the DMHP relation for some pairs of steps in the example program in Figure 1. First, let us consider \( DMHP(step_2, step_5) \). Here \( step_2 \) is to the left of \( step_5 \), since \( step_2 \) will appear before \( step_5 \) in the depth first traversal of the DPST. The lowest common ancestor of \( step_2 \) and \( step_5 \) is the node \( F_1 \). The node \( A_1 \) is the ancestor of \( step_2 \) (the left node) that is the child of \( F_1 \). Since \( A_1 \) is an async node, \( DMHP(step_2, step_5) \) will evaluate to true indicating that \( step_2 \) and \( step_5 \) can execute in parallel. This is indeed true for this program: \( step_2 \) is within \( A_1 \), while \( step_5 \) follows \( A_1 \) and is within \( A_1 \)’s immediately enclosing finish.

Now, let us consider \( DMHP(step_6, step_5) \). Here \( step_5 \) is to the left of \( step_6 \), since \( step_5 \) will appear before \( step_6 \) in the depth first traversal of the DPST. Their lowest common ancestor is \( F_1 \), and the ancestor of \( step_5 \) which is the child of \( F_1 \) is \( step_5 \) itself. Since \( step_5 \) is not an async instance, \( DMHP(step_6, step_5) \) evaluates to false. This is consistent with the program because \( step_6 \) is in task \( A_3 \) and \( A_3 \) is created only after \( step_5 \) completes.

4. Race Detection Algorithm

Our race detection algorithm involves executing the given program with a given input and monitoring every dynamic memory access in the program for potential data races. The algorithm maintains a DPST as described in the previous section, as well as the relevant access history for each shared memory location. The algorithm performs two types of actions:

- Task actions: these involve updating the DPST with a new node for each async, finish, and step instance.
- Memory actions: on every shared memory access, the algorithm checks if the access conflicts with the access history for the relevant memory location. If a conflict is detected, the algorithm reports a race and halts. Otherwise, the memory location is updated to include the memory access in its access history.

A key novelty of our algorithm is that it requires constant space to store the access history of a memory location, while still guaranteeing that no data races are missed. We next describe the shadow memory mechanism that supports this constant space guarantee.

4.1 Shadow Memory

Our algorithm maintains a shadow memory \( M_s \) for every monitored memory location \( M \). \( M_s \) is designed to store the relevant parts of the access history to \( M \). It contains the following three fields, which are all initialized to null:

- \( w \) : a reference to a step that wrote \( M \).
- \( r_s \) : a reference to a step that read \( M \).
The following invariants are maintained throughout the execution of the program until the first data race is detected.

- \(r_2\): a reference to another step that read \(M\).

- \(M_s.w\) refers to the step that last wrote \(M\).

- \(M_s.r_1\) & \(M_s.r_2\) refer to the steps that last read \(M\). All the steps \((a_1, a_2, ..., a_k)\) that have read \(M\) since the last synchronization are in the subtree rooted at \(LCA(M_s.r_1, M_s.r_2)\).

The fields of the shadow memory \(M_s\) are updated atomically by different tasks that access \(M\).

4.2 Algorithm

The most important aspect of our algorithm is that it stores only three fields for every monitored memory location irrespective of the number of steps that access that memory location. The intuition behind this is as follows: it is only necessary to store the last write to a memory location because all the writes before the last one must have completed at the end of the last synchronization. This is assuming no data races have been observed yet during the execution. Note that though synchronization due to finish may not be global, two writes to a memory location have to be ordered by execution. Note that though synchronization due to finish may not be global, two writes to a memory location have to be ordered by execution. Note that though synchronization due to finish may not be global, two writes to a memory location have to be ordered by execution.

Algorithm 1: Write Check

```
Input: Memory location \(M\), Step \(S\) that writes to \(M\)
1 if DMHP(M_s, r_1, S) then
2 Report a read-write data race between \(M_s.r_1\) and \(S\)
3 end
4 if DMHP(M_s, r_2, S) then
5 Report a read-write data race between \(M_s.r_2\) and \(S\)
6 end
7 if DMHP(M_s, w, S) then
8 Report a write-write data race between \(M_s.w\) and \(S\)
9 else
10 \(M_s.w\) ← \(S\)
11 end
```

When a step \(S\) reads a memory location \(M\), Algorithm 2 reports a write-read data race between \(M_s.w\) and \(S\) if these two steps can execute in parallel. Then, it updates the reader fields of \(M_s\) as follows: if \(S\) can never execute in parallel with either of the two readers, \(M_s.r_1\) and \(M_s.r_2\), then both these readers are discarded and \(M_s.r_1\) is set to \(S\). If \(S\) can execute in parallel with both the readers, \(M_s.r_1\) and \(M_s.r_2\), then the algorithm stores two of these three steps, whose LCA is the highest in the DPST, i.e., if \(LCA(M_s.r_1, S)\) or \(LCA(M_s.r_2, S)\) is dpst-greater than \(LCA(M_s.r_1, M_s.r_2)\), then \(M_s.r_1\) is set to \(S\). Note that in this case \(S\) is outside the subtree under \(LCA(M_s.r_1, M_s.r_2)\) and hence, \(LCA(M_s.r_1, S)\) will be the same as \(LCA(M_s.r_2, S)\).

Algorithm 2: Read Check

```
Input: Memory location \(M\), Step \(S\) that reads \(M\)
1 if DMHP(M_s, w, S) then
2 Report a write-read data race between \(M_s.w\) and \(S\)
3 end
4 if ¬DMHP(M_s, r_1, S) ∧ ¬DMHP(M_s, r_2, S) then
5 \(M_s.r_1\) ← \(S\)
6 \(M_s.r_2\) ← null
7 else if DMHP(M_s, r_1, S) ∧ DMHP(M_s, r_2, S) then
8 \(lca_{12}\) ← LCA(M_s.r_1, M_s.r_2)
9 \(lca_{1}\) ← LCA(M_s.r_1, S)
10 \(lca_{2}\) ← LCA(M_s.r_2, S)
11 if \(lca_{1} >\ dpst \land lca_{2} >\ dpst \land lca_{12}\) then
12 \(M_s.r_1\) ← \(S\)
13 end
14 end
```

Theorem 2. If Algorithms 1 and 2 do not report any data race in some execution of a program \(P\) with input \(\psi\), then no execution of \(P\) with \(\psi\) will have a data race on any memory location \(M\).

Proof. Please refer to [25].

Theorem 3. If Algorithm 1 or 2 reports a data race on a memory location \(M\) during an execution of a program \(P\) with input \(\psi\), then there exists at least one execution of \(P\) with \(\psi\) in which this race exists.

Proof. Please refer to [25].

Theorem 4. The race detection algorithm is sound and precise for a given input.

Proof. From Theorem 2 it follows that our race detection algorithm is sound for a given input. From Theorem 3 it follows that our race detection algorithm is precise for a given input.
5. Implementation and Optimizations

This section describes the implementation of the different parts of our data race detection algorithm.

5.1 DPST

The DPST of the program being executed is built to maintain the parent-child relationship of asyncs, finishes and steps in the program. Every node in the DPST consists of the following 4 fields:

- **parent**: the DPST node which is the parent of this node.
- **depth**: an integer that stores the depth of this node. The root node of the DPST has depth 0. Every other node in the DPST has depth one greater than its parent. This field is immutable.
- **num_children**: number of children of this node currently in the DPST. This field is initialized to 0 and incremented when child nodes are added.
- **seq_no**: an integer that stores the ordering of this node among the children of its parent, i.e., among its siblings. Every node’s children are ordered from left to right. They are assigned sequence numbers starting from 1 to indicate this order. This field is also immutable.

The use of depth for nodes in the DPST leads to a lowest common ancestor (LCA) algorithm with better complexity (than if we had not used this field). The use of sequence numbers to maintain the order of a node’s children makes it easier to check for may happen in parallel given two steps in the program.

Note that all the fields of a node in the DPST can be initialized updated without any synchronization: the parent field initialization is trivial because there are no competing writes to that field; the depth field of a node is written only on initialization, is never updated, and is read only after the node is created; the num_children field is incremented whenever a child node is added, but for a given node, its children are always added sequentially in order from left to right; the seq_no field is written only on initialization, is never updated, and is read only after the node is created.

5.2 Computing DMHP

A large part of the data race detection algorithm involves checking DMHP for two steps in the program. This requires computing the Lowest Common Ancestor (LCA) of two nodes in a tree. The function LCA(Γ, S1, S2) returns the lowest common ancestor of the nodes S1 and S2 in the DPST Γ. This is implemented by starting from the node with the greater depth (say S1) and traversing up Γ until a node with the same depth as S2 is reached. From that point, Γ is traversed along both the paths until a common node is reached. This common node is the lowest common ancestor of S1 and S2. The time overhead of this algorithm is linear in the length of the longer of the two paths, S1 → L and S2 → L.

Algorithm 3 computes DMHP relation between two steps S1 and S2. Algorithm 3 returns true if the given two steps S1 and S2 may happen in parallel and false otherwise. This algorithm first computes the lowest common ancestor L of the given two steps using the LCA function. If the step S1 is to the left of S2, then the algorithm returns true if the ancestor of S1 (which is the child of L) is an async and false otherwise. If the step S2 is to the left of S1, then the algorithm returns true if the ancestor of S2 which is the child of L is an async and false otherwise. The time overhead of this algorithm is same as that of the LCA function, since it only takes constant time to find the node which is the ancestor of the left step that is the child of LCA node and then check if that node is an async.

<table>
<thead>
<tr>
<th>Algorithm 3: Dynamic May Happen in Parallel (DMHP)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong>: DPST Γ, Step S1, Step S2</td>
</tr>
<tr>
<td><strong>Output</strong>: true/false</td>
</tr>
<tr>
<td>N_{LCA} = LCA(Γ, S1, S2)</td>
</tr>
<tr>
<td>A1 = Ancestor of S1 in Γ which is the child of N_{LCA}</td>
</tr>
<tr>
<td>A2 = Ancestor of S2 in Γ which is the child of N_{LCA}</td>
</tr>
<tr>
<td>if A1 is to the left of A2 in Γ then</td>
</tr>
<tr>
<td>if A1 is an Async then</td>
</tr>
<tr>
<td>return true</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>return false // S1 happens before S2</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>return false // S2 happens before S1</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

5.3 Space and Time Overhead

The size of the DPST will be O(n), where n is the number of tasks in the program. More precisely, the total number of nodes in the DPST will be 3 * (a + f) - 1, where a is the number of async instances and f is the number of finish instances in the program. This is because a program with just one finish node will have just one step node inside the finish of its DPST. When an async or a finish node is subsequently added to the DPST, it will result in adding 2 steps nodes, one as the child of the new node and the other as its sibling. The space overhead for every memory location is O(1), since we only need to store a writer step and two reader steps in the shadow memory of every memory location.

The time overhead at task boundaries is O(1), which is the time needed to add/update a node in the DPST. The worst case time overhead on every memory access is same as that of Algorithm 3.

Note that the time overhead is not proportional to the number of processors (underlying worker threads) that the program runs on. Hence, the overhead is not expected to scale as we increase the number of processors on which the program executes. This is an important property as future hardware will likely have many cores.

5.4 Relaxing the Atomicity Requirement

A memory action for an access to a memory location M involves reading the fields of its shadow memory location M_s, computing the necessary DMHP information and checking appropriate predicates, and possibly updating the fields of M_s. Let us refer to these three stages as read, compute, and update of a memory action.

In our algorithm, every memory action on a shadow memory M_s has to execute atomically relative to other memory actions on M_s. When there are parallel reads to a memory location, this atomicity requirement effectively serializes the memory actions due to these reads. Hence this atomicity requirement induces a bottleneck in our algorithm when the program is executed on a large number of threads. Note that the atomicity requirement does not result in a bottleneck in the case of writes to a memory location because the memory actions due to writes have no contention in data race free programs. (In a data race free program, there is a happens-before ordering between a write and every other access to a memory location.)

We now present our implementation strategy to overcome this atomicity requirement without sacrificing the correctness of our algorithm. This implementation strategy is based on the solution to the reader-writer problem proposed by Leslie Lamport in [19].
Our implementation allows multiple memory actions on the same shadow memory to proceed in parallel. This is done by adding two atomic integers to every shadow memory, i.e., \( M_s \) contains the following two additional fields:

- \( \text{startVersion} \): an atomic integer that denotes the version number of \( M_s \).
- \( \text{endVersion} \): an atomic integer that denotes the version number of \( M_s \).

Both \( \text{startVersion} \) and \( \text{endVersion} \) are initialized to zero. Every time any of the fields \( M_s.w, M_s.r_1, \) or \( M_s.r_2 \) is updated, \( M_s.\text{startVersion} \) and \( M_s.\text{endVersion} \) are incremented by one. The following invariant is maintained on every shadow memory \( M_s \) during the execution of our algorithm: any consistent snapshot of \( M_s \) will have the same version number in both \( \text{startVersion} \) and \( \text{endVersion} \). Now, we show how the read, compute, and update stages of a memory action on \( M_s \) are performed. Note that these rules use a \( \text{CompareAndSet} \) (CAS) primitive which is atomic relative to every operation on the same memory location.

**Read:**
- Read the version number in \( M_s.\text{startVersion} \) into a local variable, \( X \).
- Read the fields \( M_s.w, M_s.r_1, \) and \( M_s.r_2 \) into local variables, \( W, R_1, \) and \( R_2 \).
- Perform a \( \text{fence} \) to ensure that all operations above are complete.
- Read the version number in \( M_s.\text{endVersion} \) into a local variable, \( Y \).
- If \( X \) is not the same as \( Y \), restart the read stage.

**Compute:**
- Perform the computation on the local variables, \( W, R_1, \) and \( R_2 \).

**Update:**
- Do the following steps if an update to any of the fields \( M_s.w, M_s.r_1, \) or \( M_s.r_2 \) is necessary.
- Perform a CAS on the version number in \( M_s.\text{endVersion} \) looking for the value \( X \) and updating it with an increment of one.
- If the above CAS fails, restart the memory action from the beginning of the read stage.
- Write to the required fields of \( M_s \).
- Write the incremented version number to \( M_s.\text{startVersion} \).

When a memory action on \( M_s \) completes the read stage, the above rules ensure that a consistent snapshot of \( M_s \) was captured. This is because the read stage completes only when the same version number is seen in both \( M_s.\text{startVersion} \) and \( M_s.\text{endVersion} \).

The CAS in the update stage of the memory action on \( M_s \) succeeds only when \( M_s.\text{endVersion} \) has the version number that was found in the read stage earlier. The update stage completes by writing to the reader and writer fields of \( M_s \) as necessary, followed by incrementing the version number in \( M_s.\text{startVersion} \). When the update stage completes, both \( M_s.\text{startVersion} \) and \( M_s.\text{endVersion} \) will have the same version number and thus, the fields of \( M_s \) are retained in a consistent state.

The CAS in the update stage of a memory action \( \alpha \) on \( M_s \) also ensures that the fields of \( M_s \) are updated only if it has not already been updated by any memory action on \( M_s \), since the read stage of \( \alpha \). If this CAS fails, then there has been some update to \( M_s \) since the read stage and hence, the computations are discarded and the memory action is restarted from the beginning of the read stage. Thus, the memory actions are guaranteed to be atomic relative to other memory actions on the same memory location.

The main advantage of this implementation is that it allows multiple memory actions on the same shadow memory \( M_s \) to proceed in parallel. But if more than one of them needs to update the fields of \( M_s \), then only one of them is guaranteed to succeed while the others repeat the action. This is especially beneficial when there are multiple parallel accesses to \( M \) whose memory actions do not update the fields of \( M_s \). In our algorithm, this occurs when there are reads by step \( S \) such that \( S \) is in the subtree rooted at \( \text{LCA}(M_s.r_3, M_s.r_2) \). These cases occur frequently in practice thereby emphasizing the importance of relaxing the atomicity requirement.

Our algorithm is implemented in Java and we use the \( \text{AtomicInteger} \) from Java Concurrency Utilities for the version numbers. The \( \text{CAS on AtomicInteger} \) is guaranteed to execute atomically with respect to other operations on the same location. Also, the CAS acts as a barrier for the memory effects of the instructions on its either side, i.e., all the instructions above it are guaranteed to complete before it executes and no instructions below it will execute before it completes. This is the same as the memory effects of the fence that is used in the read stage. The read of an \( \text{AtomicInteger} \) has the memory effects of the read of a volatile in Java. Hence, it does not allow any instruction after it to execute until it completes. Similarly, the write to an \( \text{AtomicInteger} \) has the memory effects of the write to a volatile in Java. Hence, it does not execute until all the instructions before it complete.

### 5.5 Optimizations

In the implementation of our algorithm, we also include the static optimizations that were described in [24]. These optimizations eliminate redundant updates to the shadow memory location due to redundant reads and writes to the corresponding memory location with a single step. These are static optimizations that perform data flow analysis on the input program to identify redundant shadow memory updates. The optimizations include: main-task check elimination, read-only check elimination, escape analysis to eliminate task-local checks, loop-invariant check optimizations, and read/write check elimination. We note that these optimizations can be used to improve the performance of any race detection algorithm.

We have also identified a number of dynamic optimizations that can reduce the space and time overhead of the DMHP algorithm even further. We leave those as future work.

### 6. Experimental Results

In this section, we present experimental results for our algorithm, which for convenience we refer to as \( \text{SPD3 (Scalable Precise Dynamic Datrace Detection)} \). The algorithm was implemented as a Java library for detecting data races in HJ programs containing async and finish constructs [6]. Shadow locations were implemented by extending the \( \text{hj.lang.Object} \) class with shadow fields, and by using \( \text{array views} \) [6, 24] as anchors for shadow arrays. Programs were instrumented for race detection during a bytecode-level transformation pass implemented on HJ’s Parallel Intermediate Representation (PIR) [31]. The PIR is an intermediate representation that extends Soot’s Jimple IR [29] with parallel constructs such as async and finish. The instrumentation pass adds the necessary calls to our race detector library at async and finish boundaries and on reads and writes to shared memory locations.

We also compare \( \text{SPD3} \) with some race detectors from past work, namely Eraser [26], FastTrack [13], and ESP-bags [24]. For Eraser and FastTrack, we use the implementations included in the RoadRunner tool [14]. Since the performance of the FastTrack implementation available in the public RoadRunner download yielded worse results than those described in [13], we communicated with...
the implementers and received an improved implementation of FastTrack which was used to obtain the results reported in this paper. For ESP-bags, we used the same implementation that was used in [24].

Our experiments were conducted on a 16-core (quad-socket, quad-core per socket) Intel Xeon 2.4GHz system with 30 GB memory, running Red Hat Linux (RHEL 5), and Sun Hotspot JDK 1.6. To reduce the impact of JIT compilation, garbage collection and other JVM services, we report the smallest time measured in 3 runs repeated in the same JVM instance for each data point. HJ tasks are scheduled on a fixed number of worker threads using a work-stealing scheduler with an adaptive policy [17].

6.1 Evaluation of SPD3

We evaluated SPD3 on a suite of 15 task-parallel benchmarks listed in Table 1. It includes eight Java Grande Forum benchmarks (JGF) [28], four Barcelona OpenMP Task Suites benchmarks (BOTS) [11], two Shootout benchmarks [2], and one EC2 challenge benchmark.

All benchmarks were written using only finish and async constructs for parallelism, with fine grained one-async-per-iteration parallelism for parallel loops. As discussed later, the original version of the JGF benchmarks contained “chunked” parallel loops with programmer-specified decomposition into coarse grained one-chunk-per-thread parallelism. The fine grained task-parallel versions of the JGF benchmarks used for the evaluation in this section were obtained by rewriting the chunked loops into “unchunked” parallel loops. In addition, barrier operations in the JGF benchmarks were replaced by appropriate finish constructs.

HJ-Base refers to the uninstrumented baseline version of each of these benchmarks. All the JGF benchmarks were configured to run with the largest available input size. All input sizes are shown in Table 1.

No data race was expected in these 15 programs, and SPD3 found only one data race which turned out to be a benign race. This was due to repeated parallel assignments of the same value to the same location in the async-finish version of the MonteCarlo benchmark, which was corrected by removing the redundant assignments. After that, all the benchmarks used in this section were observed to be data-race-free for the inputs used.

Figure 3 shows the relative slowdown of SPD3 for all benchmarks when executed with 1, 2, 4, 8, and 16 worker threads. (Recall that these benchmarks create many more async tasks than the number of worker threads.) The relative slowdown on n threads refer to the slowdown of the SPD3 instrumented version of the benchmark executing on n threads compared with the HJ-Base version executing on n threads. Ideally, a scalable race detector should have a constant relative slowdown as the number of worker threads increases. As evident from Figure 3, the slowdown for many of the benchmarks decrease as the number of worker threads increases from 1 to 16. The geometric mean of the slowdowns for all the benchmarks on 16 threads is 2.78×.

Though the geometric mean is below 3×, four of the 15 benchmarks (Crypt, LUFact, RayTracer, and FFT) exhibited a slowdown around 10× for worker threads from 1 to 16. This is because these benchmarks contain larger numbers of shared locations that need to be monitored for race detection. As discussed later, other race detection algorithms exhibit much larger slowdowns for these examples than SPD3. Note that even in these cases the slowdowns are similar across 1 to 16 threads. This clearly shows that SPD3 scales well.

The slowdown for 1-thread is higher than that for all other threads in many benchmarks. This is because our implementation

---

**Table 1. List of Benchmarks Evaluated**

<table>
<thead>
<tr>
<th>Source</th>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JGF</td>
<td>Series (C)</td>
<td>Fourier coefficient analysis</td>
</tr>
<tr>
<td></td>
<td>LUFact (C)</td>
<td>LU Factorisation</td>
</tr>
<tr>
<td></td>
<td>SOR (C)</td>
<td>Successive over-relaxation</td>
</tr>
<tr>
<td></td>
<td>Crypt (C)</td>
<td>IDEA encryption</td>
</tr>
<tr>
<td></td>
<td>Sparse (C)</td>
<td>Sparse Matrix multiplication</td>
</tr>
<tr>
<td>Bots</td>
<td>MonteCarlo (B)</td>
<td>Molecular Dynamics simulation</td>
</tr>
<tr>
<td></td>
<td>RayTracer (B)</td>
<td>Monte Carlo simulation</td>
</tr>
<tr>
<td></td>
<td>FFT (large)</td>
<td>Fast Fourier Transformation</td>
</tr>
<tr>
<td></td>
<td>Health (large)</td>
<td>Simulates a country health system</td>
</tr>
<tr>
<td></td>
<td>NQueens (14)</td>
<td>N Queens problem</td>
</tr>
<tr>
<td></td>
<td>Strassen (large)</td>
<td>Matrix Multiply with Strassen’s method</td>
</tr>
<tr>
<td>Shootout</td>
<td>Fannkuch (10M)</td>
<td>Indexed-access to tiny integer-sequence</td>
</tr>
<tr>
<td></td>
<td>Mandelbrot (8000)</td>
<td>Generate Mandelbrot set portable bitmap</td>
</tr>
<tr>
<td>EC2</td>
<td>Matlab (1000’2)</td>
<td>Matrix Multiplication (Iterative)</td>
</tr>
</tbody>
</table>
uses `compareAndSet` operations on atomic variables. These operations are not optimized for the no contention scenario as with 1-thread. Instead, if we use a lock that is optimized for no contention scenario, the slowdown for 1-thread cases would have been a lot lower. But that implementation does not scale well for larger numbers of threads. For example, the lock based implementation is 1.8× slower (on average) than the `compareAndSet` implementation when running on 16-threads. While the two implementations are close for many benchmarks (within a factor of 2), there is a difference of upto 7× for some benchmarks, when running on 16-threads. The `compareAndSet` implementation is always faster than the lock based implementation for larger numbers of threads. Since our aim was to make the algorithm scalable, we chose the `compare-AndSet` approach.

### 6.2 Comparison with ESP-bags algorithm

In this section, we compare the performance of `SPD3` with ESP-bags [24]. Figure 4 shows the slowdown of ESP-bags and `SPD3` for all the benchmarks, relative to the execution time of the 16-thread HJ-Base version. Note that the ESP-bags version runs on 1-thread (because it is a sequential algorithm) while the `SPD3` version runs on 16-threads.

This comparison underscores the fact that the slowdown for a sequential approach to data race detection can be significantly larger than that of parallel approaches, when running on a parallel machine. For example, the slowdown is reduced by more than a 15× factor when moving from ESP-bags to `SPD3` for Series and MatMul benchmarks and by more than a 5× factor for benchmarks like MolDyn and SparseMatMult that scale well. On the other hand, the slowdown for Crypt is similar for ESP-bags and `SPD3` because the uninstrumented async-finish version of Crypt does not scale well. On average, `SPD3` is 3.2× faster than ESP-bags on our 16-way system. This gap is expected to further increase on systems with larger numbers of cores.

### 6.3 Comparison with Eraser and FastTrack

We only use the JGF benchmarks for comparisons with other algorithms since those are the only common benchmarks with past work on Eraser and FastTrack. However, since Eraser and FastTrack work on multithreaded Java programs rather than task-parallel vari-

![Figure 4. Slowdown of ESP-bags and `SPD3` relative to 16-thread HJ-Base version for all benchmarks. Note that the ESP-bags version runs on 1-thread while the `SPD3` version runs on 16-threads.](image)

- `ESP-bags` versus `SPD3` shows significant differences, indicating superior performance of `SPD3` for most benchmarks.
- The Y-axis represents the slowdown relative to 16-thread execution, highlighting the efficiency improvements of `SPD3`.

In summary, `SPD3` demonstrates superior performance compared to `ESP-bags` and other algorithms, making it a viable choice for data race detection in multithreaded Java programs.
We confirmed that all races were reported with the “-nobarrier” option. However, all RoadRunner performance measurements reported in this paper were obtained with default settings i.e., without the “-nobarrier” option.

Our third observation is that Eraser reported false data races for many benchmarks. This is not surprising since Eraser is known to not be a precise datarace detection algorithm.

To undertake a performance comparison, we converted the four benchmarks to race-free HJ programs by replacing the buggy barriers by finish operations. In some cases, this caused the HJ-base version to be slower than the RR-base version as a result (since RR-base measures the performance of the unmodified JGF benchmarks with custom barriers). Before we present the comparison, it is also worth noting that the implementation of Eraser and FastTrack in RoadRunner include some optimizations that are orthogonal to the race detection algorithm used [14]. Similarly, the static optimizations from [24] included in our implementation of SPD3 are also orthogonal to the race detection algorithm. Both these sets of optimizations could be performed on any race detection algorithm to improve its performance.

Table 2. Relative slowdown of Eraser, FastTrack and SPD3 for JGF benchmarks on 16 threads. The slowdown of Eraser and FastTrack was calculated over their baseline RR-Base while the slowdown of SPD3 was calculated over its baseline HJ-Base. For benchmarks marked with *, race-free versions were used for SPD3 but the original versions were used for Eraser and FastTrack.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>RR-Base Time(s)</th>
<th>Eraser</th>
<th>FastTrack</th>
<th>SPD3</th>
<th>HJ-Base Time(s)</th>
<th>SPD3 Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crypt</td>
<td>0.362</td>
<td>122.40</td>
<td>133.24</td>
<td>0.585</td>
<td>1.30</td>
<td></td>
</tr>
<tr>
<td>LUFact*</td>
<td>1.47</td>
<td>19.75</td>
<td>26.41</td>
<td>5.411</td>
<td>1.08</td>
<td></td>
</tr>
<tr>
<td>MolDyn*</td>
<td>16.185</td>
<td>8.39</td>
<td>9.59</td>
<td>3.750</td>
<td>13.56</td>
<td></td>
</tr>
<tr>
<td>MonteCarlo</td>
<td>2.878</td>
<td>10.95</td>
<td>13.54</td>
<td>5.605</td>
<td>1.66</td>
<td></td>
</tr>
<tr>
<td>RayTracer*</td>
<td>2.186</td>
<td>20.23</td>
<td>17.45</td>
<td>19.974</td>
<td>5.84</td>
<td></td>
</tr>
<tr>
<td>Series</td>
<td>112.515</td>
<td>1.00</td>
<td>1.00</td>
<td>88.768</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>SOR*</td>
<td>0.914</td>
<td>4.26</td>
<td>8.36</td>
<td>2.604</td>
<td>4.53</td>
<td></td>
</tr>
<tr>
<td>Sparse</td>
<td>2.746</td>
<td>14.29</td>
<td>20.59</td>
<td>4.607</td>
<td>1.72</td>
<td></td>
</tr>
<tr>
<td>GeoMean</td>
<td>11.23</td>
<td>13.87</td>
<td></td>
<td></td>
<td>2.63</td>
<td></td>
</tr>
</tbody>
</table>

Table 2 shows the slowdowns of Eraser, FastTrack, and SPD3 for all the JGF benchmarks on 16 threads. Note that the slowdown of Eraser and FastTrack were calculated relative to RR-Base (with 16 threads), and the slowdown of SPD3 was calculated over HJ-Base (with 16 threads). For benchmarks marked with *, race-free versions were used for SPD3 but the original versions were used for Eraser and FastTrack; this accounts for differences in the execution times of RR-Base and HJ-Base for some benchmarks since the async-finish versions include more synchronization to correct the bugs in the original Java versions.

Table 2 shows that the relative slowdowns for Eraser and FastTrack are much larger than those for SPD3. On average (geometric mean), the slowdown for SPD3 relative to HJ-base is 2.70× while that for Eraser and FastTrack are 11.21× and 13.87× respectively relative to RR-base. There is also a large variation. While the slowdowns are within a factor of 2 for SOR, there is more than a 60× gap in slowwowns for Crypt and quite a significant difference for LUFact, MonteCarlo, and SparseMatMult as well. The slowdown for SPD3 on MolDyn is larger than the slowdowns for Eraser and FastTrack because the baseline for SPD3 is more than 4× faster than the baseline for Eraser and FastTrack. For FastTrack, these slowdowns are consistent with the fact that certain data access patterns (notably, shared reads) can lead to large overheads because they prevent the use of optimized versions of vector clocks.

For the case with the largest gap in Table 2 (Crypt), Figure 5 shows the slowdown (scaled execution time) of RR-Base, Eraser, FastTrack, HJ-Base, and SPD3 for the chunked version of the Crypt benchmark on 1-16 threads relative to the 16-thread RR-Base execution time. In this benchmark, RR-Base is the fastest for 16 threads as expected. The execution time of HJ-base is 1.9× slower than RR-Base in the 1-thread case and 1.6× slower than RR-Base in the 16-thread case. Similarly, the execution time of SPD3 version is also very close; it is 4.2× slower in the 1-thread case and 3× slower in the 16-thread case. The execution time of Eraser and FastTrack are 13.7× and 16.6× slower than RR-Base in the 1-thread case but they increase to more than 100× for 8-threads and 16-threads. This example shows that for some programs the performance overheads for Eraser and FastTrack can increase dramatically with the number of threads (cores).

6.4 Memory Overhead

We now compare the memory overheads of the Eraser, FastTrack and SPD3 algorithms on the coarse-grained JGF benchmarks. Again, the baseline for Eraser and FastTrack was RR-Base and the baseline for SPD3 was HJ-Base. To obtain a coarse estimation of the memory used, we used the -verbose:gc option in the JVM and picked the maximum heap memory used over all the GC executions in a single JVM instance. All three instrumented versions trigger GC frequently, so this is a reasonable estimate of the memory overhead.

Table 3. Peak heap memory usage of RR-Base, Eraser, FastTrack, HJ-Base, and SPD3 for JGF benchmarks on 16 threads. For benchmarks marked with *, race-free versions were used for SPD3 but the original versions were used for Eraser and FastTrack.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>RR-Base Memory (in MB)</th>
<th>Eraser Memory (in MB)</th>
<th>FastTrack Memory (in MB)</th>
<th>HJ-Base Memory (in MB)</th>
<th>SPD3 Memory (in MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crypt</td>
<td>209</td>
<td>8539</td>
<td>8535</td>
<td>149</td>
<td>6009</td>
</tr>
<tr>
<td>LUFact</td>
<td>80</td>
<td>1790</td>
<td>2455</td>
<td>47</td>
<td>203</td>
</tr>
<tr>
<td>MolDyn</td>
<td>382</td>
<td>1048</td>
<td>1040</td>
<td>9</td>
<td>35</td>
</tr>
<tr>
<td>MonteCarlo</td>
<td>1771</td>
<td>9316</td>
<td>9292</td>
<td>557</td>
<td>584</td>
</tr>
<tr>
<td>RayTracer</td>
<td>1106</td>
<td>4475</td>
<td>4466</td>
<td>43</td>
<td>88</td>
</tr>
<tr>
<td>Series</td>
<td>80</td>
<td>1067</td>
<td>1062</td>
<td>162</td>
<td>177</td>
</tr>
<tr>
<td>SOR</td>
<td>81</td>
<td>1161</td>
<td>1551</td>
<td>47</td>
<td>202</td>
</tr>
<tr>
<td>Sparse</td>
<td>225</td>
<td>2120</td>
<td>2171</td>
<td>88</td>
<td>714</td>
</tr>
</tbody>
</table>

Table 3 shows the estimated memory usage of these three algorithms and their baselines for JGF benchmarks on 16 threads. The table shows that the memory usage of HJ-Base is lower than that of RR-Base in all the benchmarks except Series. In all cases, the memory usage is lower for SPD3, compared to Eraser and FastTrack with significant variation in the gaps. The memory usage of Crypt with SPD3 is quite high because the benchmark has arrays
of size 20 million and our algorithm maintains shadow locations for all elements of these arrays. But the memory used by SPD3 for Crypt is still less than that of Eraser and FastTrack. The high memory usage for Eraser and FastTrack is not surprising because Eraser has to maintain all the locks held while accessing a particular location, and FastTrack’s vector clocks may grow linearly in the number of threads.

For one of the benchmarks in Table 3 (LUFact), Figure 6 shows the estimated memory usage of the three algorithms and their baselines as a function of the number of threads/cores used. Note that both the baselines (RR-Base and HJ-Base) are very close. While the estimated heap usage of RR-Base remains constant at $80M$, the estimated usage of HJ-Base varies from $33M$ to $47M$ as we go from 1 thread to 16 threads. The estimated heap usage of SPD3 is about $6 \times$ larger than HJ-Base; it varies between $192M$ and $203M$ across 16 threads. The estimated heap usage of Eraser increases from $833M$ for 1 thread to $1790M$ for 16 threads ($2.1 \times$ increase). Similarly, the estimated heap usage of FastTrack increases from $825M$ for 1 thread to $2455M$ for 16 threads ($3 \times$ increase). This clearly shows the increase in the memory usage for Eraser and FastTrack as we increase the number of threads for this benchmark.

7. Related Work

In the introduction, we outlined the key differences between our algorithm and FastTrack. In summary, on one hand, our algorithm uses $O(1)$ space per memory location, while in the worst-case, FastTrack uses $O(n)$. On the other, FastTrack handles more general computation graphs than those supported by our model. The time overhead of our algorithm is characteristic of the application, since it depends on the height of the LCA nodes in the DPST. It is independent of the number of threads (processors) the program executes on. On the other hand, FastTrack’s worst-case time overhead is linear in the number of threads, which can grow very large with increasing numbers of cores.

Schönberg [27] presented one of the earliest dynamic data race detection algorithms for nested fork-join and synchronization operations. In this algorithm, a shared variable set is associated with each sequential block in every task. There is also a concurrency list associated with each shared variable set which keeps track of the concurrent shared variable sets that will complete at a later time. The algorithm detects anomalies by comparing complete concurrent shared variable sets at each time step. This algorithm applies only to a single execution instance of a program, as mentioned in [27]. The space required to store read information in the shared variable sets is bounded by $V \times N$, where $V$ is the number of variables being monitored and $N$ is the number of execution threads $^1$. This space requirement increases with an increase in the number of threads the program is executed on, whereas our algorithm’s space requirement is independent of the number of threads the program is executed on. A limitation of this work is that since access anomalies are detected at synchronization points, it does not identify the actual read and write operations involved in the data races.

Offset-Span (OS) labeling [21] is an optimized version of the English-Hebrew (EH) labeling technique [10] for detecting data races. The idea behind both these techniques is to attach a label to every thread in the program and use these labels to check if two threads can execute concurrently. They also maintain the access history for every shared variable that is monitored which is then used to check for conflicts. The length of the labels associated with each thread can grow arbitrarily long in EH labeling$^2$, whereas the length of the labels in OS labeling is bounded by the maximum nesting depth of fork-join in the program. While the EH labeling technique needs an access history of size equal to the number of threads for every monitored variable in the program, the OS labeling technique only needs constant size to store access history. While OS labeling algorithm supports only nested fork-join constructs, our algorithm supports a more general set of dynamic graphs. Further, though the OS labeling algorithm can execute the input program in parallel, it has been evaluated in a sequential setting only [22]. The effectiveness of this algorithm in a parallel implementation is not clear.

A related work on data race detection for structured parallel programs was done as part of the Cilk project [4]. This work gives an algorithm called SP-hybrid, which detects races in the program with a constant space and time overhead. Their algorithm has the best possible theoretical overheads for both space and time. However, despite its good theoretical bounds, the SP-hybrid algorithm is very complex and incurs significant inefficiencies in practice. The original paper on SP-hybrid [4] provides no evaluation and subsequent evaluation of an incomplete implementation of SP-hybrid [18] was done only for a small number of processors. One indicator of the inefficiency of SP-hybrid can be seen in the fact that the CilkScreen race detector used in Intel Cilk++ [1] uses the sequential All-Sets algorithm [8] rather than the parallel SP-hybrid algorithm. Another drawback of their algorithm is that it is tightly coupled with Cilk’s work-stealing scheduler. Hence, their algorithm cannot be applied directly to other schedulers. In contrast, our algorithm is amenable to an efficient implementation, performs very well in practice, supports a more general set of computation graphs than Cilk’s spawn/sync and is also independent of the underlying scheduler.

There has also been work on data race detection algorithms for spawn/sync [12] and async/finish models [24]. While they require only $O(1)$ space overhead per memory location, these algorithms must process the program in a sequential depth-first manner, fundamentally limiting the scalability of these approaches. In contrast, the algorithm presented in this work can process the program during parallel execution, while still requiring only $O(1)$ space per memory location.

---

$^1$ If $N$ refers to the maximum number of threads possible in all executions of a program for a given input, then this algorithm can guarantee data race freedom for all executions of the program for that input. If not, then this guarantee will not hold.

$^2$ Note that the length of the labels is bounded by the maximum nesting level of fork-join in EH labeling in the presence of an effective heuristic as reported in [10]
8. Conclusion and Future Work

In this work, we presented a new dynamic data race detection algorithm for structured parallel programs. The algorithm can process the program in parallel, uses $O(1)$ space per memory location and admits an efficient implementation. The algorithm tracks what can happen in parallel via a new data structure called the dynamic program structure tree (DPST), and maintains two readers and a writer for each shared memory location in order to track potential conflicts between different tasks. We implemented the algorithm and demonstrated its effectiveness on a range of benchmarks. In future, it could be interesting to extend the algorithm to other structured parallel constructs such as HJ’s phaser construct [6].

Acknowledgments

We are grateful to the authors of the RoadRunner tool [14], Cormac Flanagan and Stephen Freund, for sharing their implementation of FastTrack that was used to obtain the results reported in [13], and for answering our questions related to both FastTrack and RoadRunner. We would also like to thank John Mellor-Crummey from Rice University for his feedback and suggestions on this work. This work was supported in part by the U.S. National Science Foundation through awards 0926127 and 0964520. We also thank the US-Israel Binational Foundation (BSF) for their support.

References


