Scalable and Precise Dynamic Datarace Detection for Structured Parallelism

Raghavan Raman
Rice University
raghav@rice.edu

Jisheng Zhao
Rice University
jisheng.zhao@rice.edu

Vivek Sarkar
Rice University
vsarkar@rice.edu

Martin Vechev
ETH Zürich
martin.vechev@inf.ethz.ch

Eran Yahav *
Technion
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× 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

* Deloro Fellow

PLDI'12, June 11–16, 2012, Beijing, China.

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-

rithm depends on a particular scheduling technique (i.e. a work-stealing scheduler).

Collectively, these three limitations raise the following question: *Is there a precise dynamic race detector that works in parallel, uses O(1) space per memory location, and is suitable for practical use?* In this paper we introduce such dynamic race detector targeting structured parallel languages such as Cilk [5], OpenMP 3.0 [23], X10 [7], and Habanero Java (HJ) [6]. Our algorithm runs in parallel, uses O(1) space per memory location, and performs well in practice.

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) [3].

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.
  - is sound and precise for a given input.
- 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 $2.78\times$ on a 16-core SMP system.

The 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.

2. Background

In this section, we give a brief overview of the structured parallel model targeted by this paper. We focus on the async/finish structured parallelism constructs used in X10 [7] and Habanero Java (HJ) [6]. The async/finish constructs generalize the traditional spawn/sync constructs used in the Cilk programming system [5] since they can express a broader set of computation graphs than those expressible with the spawn/sync constructs used in Cilk [15].

While X10 and HJ include other synchronization techniques such as futures, clocks/phasers, and Cilk even includes locks, the core task creation and termination primitives in these languages are fundamentally based on the async/finish and spawn/sync constructs. The underlying complexity of a dynamic analysis algorithm is determined by these core constructs. Once a dynamic analysis algorithm for the core constructs is developed, subsequent extensions can be built on top of the core algorithm. To underscore the importance of studying the core portions of these languages, a calculus called Featherweight X10 (FX10) was proposed [20]. Also, the SP-bags algorithm [12] for Cilk was presented for the core spawn/sync constructs (the algorithm was later extended to handle accumulators and locks [8]).

The algorithm presented in this paper is applicable to async/finish constructs (which means it also handles spawn/sync constructs). The algorithm is independent of the sequential portions of the language, meaning that one can apply it to any language where the parallelism is expressed using the async/finish constructs. For example, the sequential portion of the language can be based on the sequential portions of Java as in HJ or C/C++ as in Cilk [15], Cilk++ [1], OpenMP 3.0 [23], and Habanero-C [9]. Next, we informally describe the semantics of the core async/finish constructs. A formal operational semantics can be found in [20].

Informal Semantics The statement `async { s }` causes the parent task to create a new child task to execute `s` *asynchronously* (i.e., before, after, or in parallel) with the remainder of the parent task. The statement `finish { s }` causes the parent task to execute `s` and then wait until all async tasks created within `s` have completed, including the transitively spawned tasks. Each dynamic instance T_A of an `async` task has a unique *Immediately Enclosing Finish* (IEF) instance *F* of a finish statement during program execution, where *F* is the innermost dynamic `finish` scope containing T_A. There is an implicit `finish` scope surrounding the body of *main()* so program execution will only end after all `async` tasks have completed.

The `finish` statement is a restricted join: while in the general unstructured fork-join case, a task can join with any other task, with the `finish` statement, a task can only join on tasks that are created in the enclosed statement. This is a fundamental difference between arbitrary unstructured fork-join and the async/finish (or spawn/sync) constructs. It is such restrictions on the join that make it possible to prove the absence of deadlocks for any program in the language [20], and provide an opportunity for discovering analysis algorithms that are more efficient than those for the general unstructured fork-join case.

As mentioned earlier, async/finish constructs can express a broader set of computation graphs than Cilk's spawn/sync constructs. The key relaxation in async/finish over spawn/sync is the way a task is allowed to join with other tasks as well as dropping the requirement that a parent task must wait for all of its child tasks to terminate. With spawn/sync, at any given sync point in a task execution, the task must join with *all* of its descendant tasks (and all recursive descendant tasks, by transitivity) created in between the start of the task and the join point. In contrast, with async/finish it is possible for a task to join with *some* rather than all of its descendant tasks: at the end of a finish block, the task only waits until the

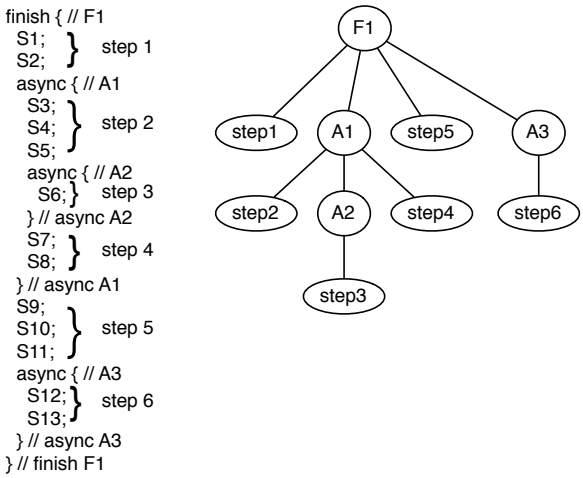

Figure 1. An example async/finish program and its final DPST.

descendant tasks created inside the finish scope have completed. More details comparing spawn/sync and async/finish can be found in [16].

Example Consider the example in Figure 1. For now, ignore the tree on the right and the step annotations, both of which are discussed in the next section. Initially, the main task begins execution with the main finish statement, labeled F1. It executes the first two statements S1 and S2 and then forks a new task A1 using the async statement. In turn, A1 executes statements S3, S4, S5 and forks task A2 which executes statement S6. Note that statement S6 (in task A2) and statements S7 and S8 in task A1 can execute in parallel. After forking A1, the main task can proceed to execute statements S9, S10 and S11 that follow A1. The main task then forks task A3 which executes statements S12 and S13. Note that the statement S11 (in the main task) and statements S12, S13 (in task A3) cannot execute in parallel because the task A3 will be forked only after the completion of S11. After forking A3, the main task has to wait until A1, A2, and A3 have terminated. Only after all these descendant tasks complete, the main task can exit past the end of finish F1.

3. Dynamic Program Structure Tree

Any dynamic data race detection algorithm needs to provide mechanisms that answer two questions: for any pair of memory accesses (with at least one write): (i) determine whether the accesses can execute in parallel, and (ii) determine whether they access the same location. In this section, we introduce the Dynamic Program Structure Tree (DPST), a data structure which can be used to answer the first question.

The DPST is an ordered rooted tree that is built at runtime to capture parent-child relationships among async, finish, and step (defined below) instances of a program. The internal nodes of a DPST represent async and finish instances. The leaf nodes of a DPST represent the steps of the program. The DPST can also be used to support dynamic analysis of structured parallel programs written in languages such as Cilk and OpenMP 3.0.

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.

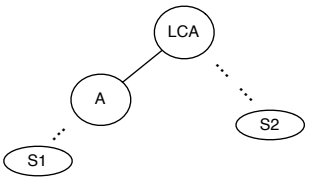

Figure 2. A part of a DPST. LCA is the Lowest Common Ancestor of steps S1 and S2. A is the DPST ancestor of S1 which is the child of LCA. S1 and S2 can execute in parallel if and only if A is an async node.

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 A1 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 n_0 is uniquely identified by a finite path from n_0 to F:

$$n_0 \xrightarrow{r_0} n_1 \xrightarrow{r_1} n_2 \xrightarrow{r_2} \ldots \xrightarrow{r_{k-1}} n_k$$

  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, S_1 and S_2, in a DPST Γ that corresponds to a program P with input ψ, may execute in parallel if and only if there exists at least one schedule δ of P with input ψ 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) S1 and S2 in a DPST, where S1 $\neq$ S2 and S1 is to the left of S2 as shown in Figure 2. Let LCA be the node denoting the least common ancestor of S1 and S2. Let node A be the ancestor of S1 that is the child of LCA. Then, S1 and S2 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(step2, step5)*. Here *step2* is to the left of *step5*, since *step2* will appear before *step5* in the depth first traversal of the DPST. The lowest common ancestor of *step2* and *step5* is the node F1. The node A1 is the ancestor of *step2* (the left node) that is the child of F1. Since A1 is an async node, *DMHP(step2, step5)* will evaluate to *true* indicating that *step2* and *step5* can execute in parallel. This is indeed *true* for this program: *step2* is within A1, while *step5* follows A1 and is within A1's immediately enclosing finish.

Now, let us consider *DMHP(step6, step5)*. Here *step5* is to the left of *step6*, since *step5* will appear before *step6* in the depth first traversal of the DPST. Their lowest common ancestor is F1, and the ancestor of *step5* which is the child of F1 is *step5* itself. Since *step5* is not an async instance, *DMHP(step6, step5)* evaluates to *false*. This is consistent with the program because *step6* is in task A3 and A3 is created only after *step5* 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_1 : a reference to a step that read M.
- r_2 : a reference to another step that read M.

The following invariants are maintained throughout the execution of the program until the first data race is detected.

- $M_s.w$ 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 some synchronization to avoid constituting a data race. Among the reads to a memory location, $(a_1, a_2, ..., a_k)$, since the last synchronization, it is only necessary to store two reads, a_i, a_j, such that the subtree under *LCA(a_i, a_j)* includes all the reads $(a_1, a_2, ..., a_k)$. This is because every future read, a_n, which is in parallel with any discarded step will also be in parallel with at least one of a_i or a_j. Thus, the algorithm will not miss any data race by discarding these steps.

Definition 5. *In a DPST, a node n_1 is dpst-greater than a node n_2, denoted by $n_1 >_{dpst} n_2$, if n_1 is an ancestor of n_2 in the DPST. Note that, in this case, n_1 is higher in the DPST (closer to the root) than n_2.*

Algorithm 1: Write Check

```
Input: Memory location M, Step S that writes to M
if DMHP(M_s.r1, S) then
    Report a read-write race between M_s.r1 and S
end
if DMHP(M_s.r2, S) then
    Report a read-write race between M_s.r2 and S
end
if DMHP(M_s.w, S) then
    Report a write-write race between M_s.w and S
else
    M_s.w ← S
end
```

Algorithms 1 and 2 show the checking that needs to be performed on write and read accesses to monitored memory locations. When a step S writes to a memory location M, Algorithm 1 checks if S may execute in parallel with the reader in $M_s.r_1$ by computing *DMHP(S, $M_s.r_1$)*. If they can execute in parallel, the algorithm reports a read-write data race between $M_s.r_1$ and S. Similarly, the algorithm reports a read-write data race between $M_s.r_2$ and S if these two steps can execute in parallel. Then, Algorithm 1 reports a write-write data race between $M_s.w$ and S, if these two steps can execute in parallel. Finally, it updates the writer field, $M_s.w$, with the current step S indicating the latest write to M. Note that this happens only when the write to M by S does not result in data race with any previous access to M.

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

Algorithm 2: Read Check

```
Input: Memory location M, Step S that reads M
if DMHP(M_s.w, S) then
    Report a write-read data race between M_s.w and S
end
if ¬DMHP(M_s.r1, S) ∧ ¬DMHP(M_s.r2, S) then
    M_s.r1 ← S
    M_s.r2 ← null
else if DMHP(M_s.r1, S) ∧ DMHP(M_s.r2, S) then
    lca12 ← LCA(M_s.r1, M_s.r2)
    lca1s ← LCA(M_s.r1, S)
    lca2s ← LCA(M_s.r2, S)
    if lca1s >dpst lca12 ∨ lca2s >dpst lca12 then
        M_s.r1 ← S
    end
end
```

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 the 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)*.

If S can execute in parallel with one of the two readers and not the other, then the algorithm does not update the readers because, in that case, S is guaranteed to be within the subtree under the *LCA($M_s.r_1$, $M_s.r_2$)*.

The *DMHP($M_s.r_2$, S)* can be computed from *DMHP($M_s.r_1$, S)* in some cases. This can be used to further optimize Algorithms 1 and 2. We do not present the details of this optimization here.

Atomicity Requirements A memory action for an access to a memory location M involves reading the fields of M_s, checking the predicates, and possibly updating the fields of M_s. Every such memory action has to execute atomically with respect to other memory actions for accesses to the same memory location.

Theorem 2. *If Algorithms 1 and 2 do not report any data race in some execution of a program P with input ψ, then no execution of P with ψ 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 ψ, then there exists at least one execution of P with ψ 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 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 ordering 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 $LCA(\Gamma, S_1, S_2)$ returns the lowest common ancestor of the nodes S_1 and S_2 in the DPST Γ. This is implemented by starting from the node with the greater depth (say S_1) and traversing up Γ until a node with the depth same as S_2 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 S_1 and S_2. The time overhead of this algorithm is linear in the length of the longer of the two paths, $S_1 \rightarrow L$ and $S_2 \rightarrow L$.

Algorithm 3 computes *DMHP* relation between two steps S_1 and S_2. Algorithm 3 returns *true* if the given two steps S_1 and S_2 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 S_1 is to the left of S_2, then the algorithm returns *true* if the ancestor of S_1 (which is the child of L) is an async and *false* otherwise. If the step S_2 is to the left of S_1, then the algorithm returns *true* if the ancestor of S_2 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.

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.

Algorithm 3: Dynamic May Happen in Parallel (*DMHP*)

Input: DPST Γ, Step S_1, Step S_2
Output: true/false

```
1  N_lca = LCA(Γ, S1, S2)
2  A1 = Ancestor of S1 in Γ which is the child of N_lca
3  A2 = Ancestor of S2 in Γ which is the child of N_lca
4  if A1 is to the left of A2 in Γ then
5      if A1 is an Async then
6          return true
7      else
8          return false // S1 happens before S2
9      end
10 else
11     if A2 is an Async then
12         return true
13     else
14         return false // S2 happens before S1
15     end
16 end
```

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:

- *startVersion*: an atomic integer that denotes the version number of M_s

- *endVersion*: an atomic integer that denotes the version number of M_s.

Both *startVersion* and *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.*startVersion* and M_s.*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 startVersion and 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 *CompareAndSet (CAS)* primitive which is *atomic* relative to every operation on the same memory location.

Read:

- Read the version number in M_s.*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 *fence* to ensure that all operations above are complete.
- Read the version number in M_s.*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.*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 *read* stage.
- Write to the required fields of M_s.
- Write the incremented version number to M_s.*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.*startVersion* and M_s.*endVersion*.

The *CAS* in the *update* stage of the memory action on M_s succeeds only when M_s.*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.*startVersion*. When the *update* stage completes, both M_s.*startVersion* and M_s.*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 α 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 α. 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 *LCA($M_s.r_1$, $M_s.r_2$)*. 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 *AtomicInteger* from Java Concurrency Utilities for the version numbers. The *CAS* on *Atomic Integer* 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 *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 *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 *SPD3* (Scalable Precise Dynamic Datarace 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 `hj.lang.Object` class with shadow fields, and by using *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 *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 mem-

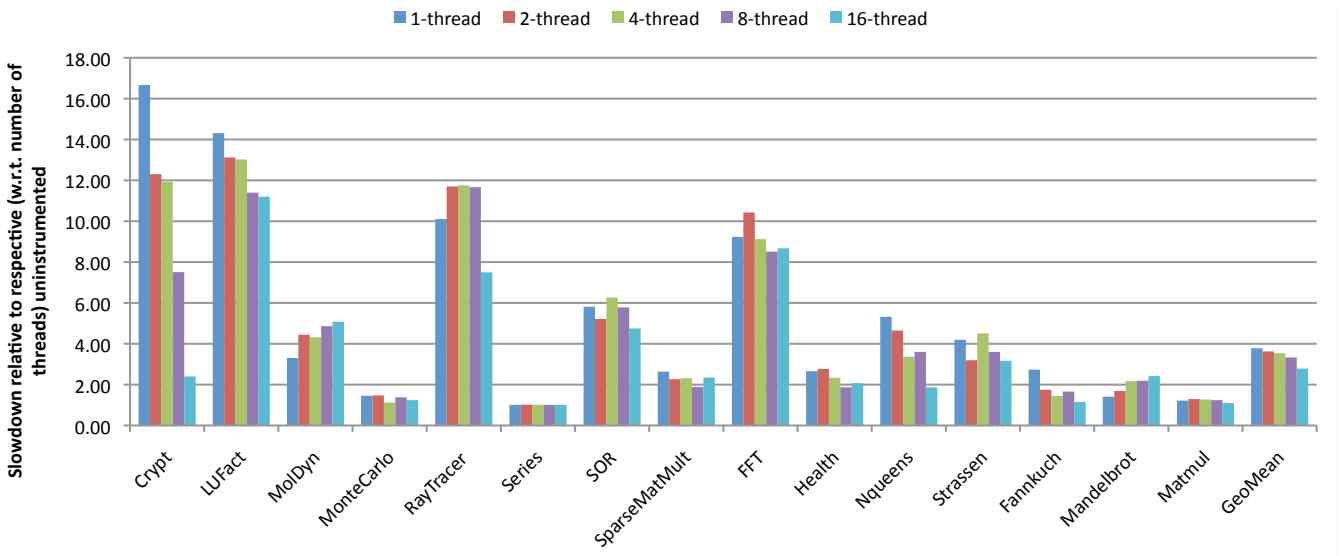

Figure 3. Relative slowdown of *SPD3* for all benchmarks on 1, 2, 4, 8, and 16 threads. Relative slowdown on *n* threads refers to the slowdown of the *SPD3* version on *n* threads compared to the HJ-Base version on *n* threads.

Table 1. List of Benchmarks Evaluated

Source	Benchmark	Description
JGF (Section 2)	Series (C)	Fourier coefficient analysis
	LUFact (C)	LU Factorisation
	SOR (C)	Successive over-relaxation
	Crypt (C)	IDEA encryption
	Sparse (C)	Sparse Matrix multiplication
JGF (Section 3)	MolDyn (B)	Molecular Dynamics simulation
	MonteCarlo (B)	Monte Carlo simulation
	RayTracer (B)	3D Ray Tracer
Bots	FFT (large)	Fast Fourier Transformation
	Health (large)	Simulates a country health system
	NQueens (14)	N Queens problem
	Strassen (large)	Matrix Multiply with Strassen's method
Shootout	Fannkuch (10M)	Indexed-access to tiny integer-sequence
	Mandelbrot (8000)	Generate Mandelbrot set portable bitmap
EC2	Matmul (1000^2)	Matrix Multiplication (Iterative)

ory, 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 $2.78\times$.

Though the geometric mean is below $3\times$, four of the 15 benchmarks (Crypt, LUFact, RayTracer, and FFT) exhibited a slowdown around $10\times$ 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 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

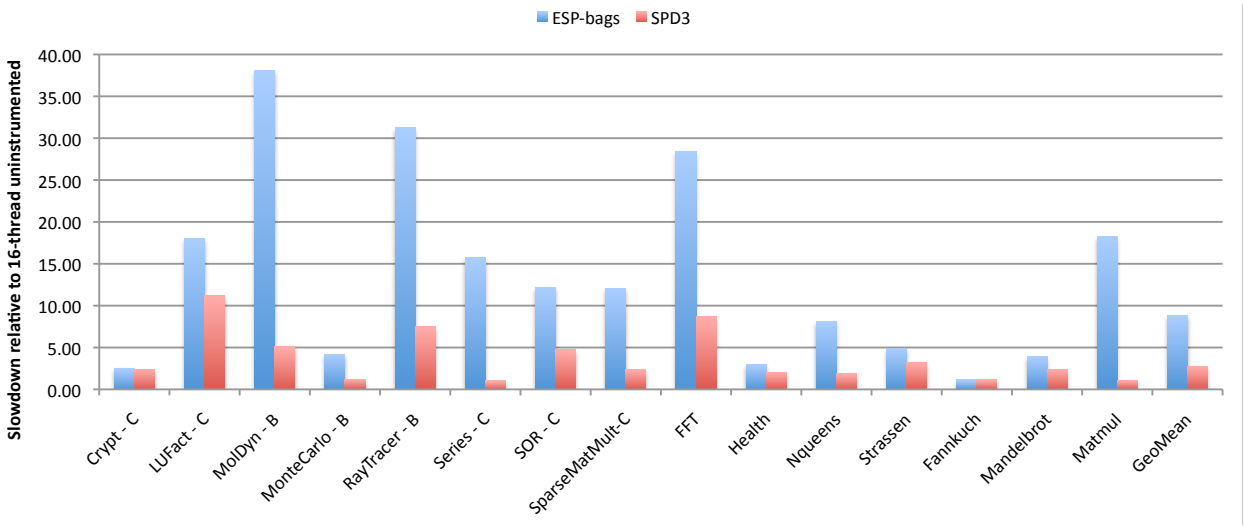

Figure 4. Slowdown of ESP-bags and *SPD3* relative to 16-thread HJ-Base version for all benchmarks. Note that the ESP-bags version runs on 1-thread while the *SPD3* version runs on 16-threads.

is $1.8\times$ 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 $7\times$ 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 *compareAndSet* 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 datarace 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 $15\times$ factor when moving from ESP-bags to *SPD3* for Series and MatMul benchmarks and by more than a $5\times$ 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 $3.2\times$ 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 variants like HJ, they used the original coarse-grained one-chunk-per-thread approach to loop parallelism in the JGF benchmarks with one thread per core. Converting these programs to fine-grained parallel versions using Java threads was not feasible since creating large numbers of threads quickly leads to OutOfMemoryError's. Further, it would also make the size of the vector clocks prohibitively large in the program in order to provide the same soundness and completeness guarantees as *SPD3*.

So, to enable an apples-to-apples comparison in this section, we created coarse-grained async-finish versions of the JGF benchmarks with chunked loops for the HJ versions. Since Eraser and FastTrack were implemented in RoadRunner, we used the execution of the Java versions of these benchmarks on RoadRunner without instrumentation (RR-Base) as the baseline for calculating the slowdowns for Eraser and FastTrack. The differences between RR-Base and HJ-Base arise from the use of array views in the HJ version, and from the use of finish operations instead of barriers as discussed below.

Our first observation when running *SPD3* on the coarse grained HJ versions of the eight JGF benchmarks was that data races were reported for four of the benchmarks: LUFact, MolDyn, RayTracer, and SOR. The data race reports pointed to races in shared arrays that were used by the programmer to implement custom barriers. However, all the custom barrier implementations were incorrect because they involved unsynchronized spin loops on shared array elements. Even though the programmer declared the array references as volatile, the volatile declaration does not apply to the elements of the array. (In all fairness to the programmer, the JGF benchmarks were written in the late 1990's when many Java practitioners were unaware of the implications of the Java memory model.)

Our second observation is that the default Eraser and FastTrack tools in the RoadRunner implementation did not report most of these data races. The only race reported was by FastTrack for SOR. After communication with the implementers of RoadRunner, we recently learned that RoadRunner recognizes a number of common barrier class implementations by default and generates special Barrier Enter and Barrier Exit events for them which in turn enables Eraser and FastTrack to take the barriers into account for race detection (even though the barriers are technically buggy). Further a "-nobarrier" option can be used to suppress this barrier detection. 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.

Benchmark	RR-Base Time(s)	Eraser Slowdown	FastTrack Slowdown	HJ-Base Time(s)	*SPD3* Slowdown
Crypt	0.362	122.40	133.24	0.585	1.84
LUFact*	1.47	17.95	26.41	5.411	1.08
MolDyn*	16.185	8.39	9.59	3.750	13.56
MonteCarlo	2.878	10.95	13.54	5.605	1.86
RayTracer*	2.186	20.23	17.45	19.974	5.84
Series	112.515	1.00	1.00	88.768	1.00
SOR*	0.914	4.26	8.36	2.604	4.53
Sparse	2.746	14.29	20.59	4.607	1.72
GeoMean	-	11.21	13.87	-	2.63

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 $2.70\times$ while that for Eraser and FastTrack are $11.21\times$ and $13.87\times$ 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 $60\times$ gap in slowdowns 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 $4\times$ 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 $1.9\times$ slower than RR-Base in the 1-thread case and $1.6\times$ slower than RR-Base in the 16-thread case. Similarly, the execution time of *SPD3* version is also very close; it is $4.2\times$ slower in the 1-thread case and $3\times$ slower in the 16-thread case. The execution time

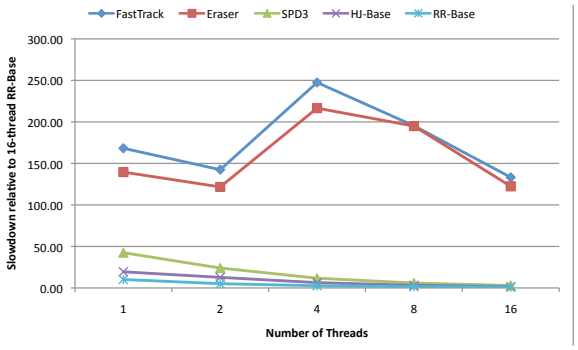

Figure 5. Slowdown (relative to 16-threads RR-Base) of RR-Base, Eraser, FastTrack, HJ-Base, and *SPD3* for Crypt benchmark (chunked version) on 1-16 threads

of Eraser and FastTrack are $13.7\times$ and $16.6\times$ slower than RR-Base in the 1-thread case but they increase to more than $100\times$ 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.

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

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 in the worst case.

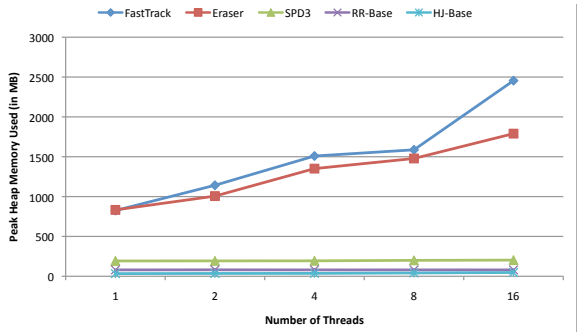

Figure 6. Estimated heap memory usage (in MB) of RR-Base, Eraser, FastTrack, HJ-Base, and *SPD3* for LUFact benchmark

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.

Schonberg [27] presented one of the earliest dynamic data race detection algorithm 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.

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,

[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]

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

[1] Intel Cilk++ Programmer's Guide. http://software.intel.com/en-us/articles/download-intel-cilk-sdk/.

[2] The Computer Language Benchmarks Game. http://shootout.alioth.debian.org/.

[3] BARIK, R., ET AL. The habanero multicore software research project. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications* (2009), ACM, pp. 735–736.

[4] BENDER, M. A., ET AL. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *SPAA'04* (Barcelona, Spain, June27–30 2004), pp. 133–144.

[5] BLUMOFE, R. D., ET AL. Cilk: an efficient multithreaded runtime system. In *PPoPP'95* (Oct. 1995), pp. 207–216.

[6] CAVE, V., ZHAO, J., SHIRAKO, J., AND SARKAR, V. Habanero-java: the new adventures of old x10. In *PPPJ'11* (2011).

[7] CHARLES, P., ET AL. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA 2005 Onward! Track* (2005).

[8] CHENG, G.-I., FENG, M., LEISERSON, C. E., RANDALL, K. H., AND STARK, A. F. Detecting data races in cilk programs that use locks. In *SPAA'98* (1998), pp. 298–309.

[9] CONG, J., SARKAR, V., REINMAN, G., AND BUI, A. Customizable Domain-Specific Computing. *IEEE Design and Test*, 2:28 (Mar 2011), 6–15.

[10] DINNING, A., AND SCHONBERG, E. An empirical comparison of monitoring algorithms for access anomaly detection. In *PPoPP'90* (1990), ACM, pp. 1–10.

[11] DURAN, A., ET AL. Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *ICPP'09* (2009), pp. 124 –131.

[12] FENG, M., AND LEISERSON, C. E. Efficient detection of determinacy races in cilk programs. In *SPAA'97* (1997), ACM, pp. 1–11.

[13] FLANAGAN, C., AND FREUND, S. N. FastTrack: efficient and precise dynamic race detection. In *PLDI '09* (2009), ACM, pp. 121–133.

[14] FLANAGAN, C., AND FREUND, S. N. The roadrunner dynamic analysis framework for concurrent programs. In *PASTE'10* (2010), ACM, pp. 1–8.

[15] FRIGO, M., LEISERSON, C. E., AND RANDALL, K. H. The implementation of the cilk-5 multithreaded language. In *PLDI'98* (1998), ACM, pp. 212–223.

[16] GUO, Y., ET AL. Work-first and help-first scheduling policies for async-finish task parallelism. In *IPDPS'09* (2009), IEEE Computer Society, pp. 1–12.

[17] GUO, Y., ZHAO, J., CAVÉ, V., AND SARKAR, V. Slaw: A scalable locality-aware adaptive work-stealing scheduler. In *IPDPS* (2010).

[18] KARUNARATNA, T. C. Nondeterminator-3: A provably good data-race detector that runs in parallel. Master's thesis, Department of Electrical Engineering and Computer Science, MIT,, Sept. 2005.

[19] LAMPORT, L. Concurrent reading and writing. *Commun. ACM 20* (November 1977), 806–811.

[20] LEE, J. K., AND PALSBERG, J. Featherweight x10: a core calculus for async-finish parallelism. In *PPoPP'10* (2010), ACM, pp. 25–36.

[21] MELLOR-CRUMMEY, J. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Supercomputing'91* (1991), ACM, pp. 24–33.

[22] MELLOR-CRUMMEY, J. Compile-time support for efficient data race detection in shared-memory parallel programs. In *PADD '93: Proceedings of the ACM/ONR workshop on Parallel and distributed debugging* (1993), ACM, pp. 129–139.

[23] OpenMP Application Program Interface v 3.0, 2008.

[24] RAMAN, R., ET AL. Efficient data race detection for async-finish parallelism. In *RV'10* (2010), Springer-Verlag, pp. 368–383.

[25] RAMAN, R., ZHAO, J., SARKAR, V., VECHEV, M., AND YAHAV, E. Scalable and precise dynamic datarace detection for structured parallelism. Tech. Rep. TR12-01, Department of Computer Science, Rice University, Houston, TX, 2012.

[26] SAVAGE, S., ET AL. Eraser: a dynamic data race detector for multi-threaded programs. *ACM Trans. Comput. Syst. 15*, 4 (1997), 391–411.

[27] SCHONBERG, E. On-the-fly detection of access anomalies. In *PLDI'98* (1998), pp. 285–297.

[28] SMITH, L. A., AND BULL, J. M. A Parallel Java Grande Benchmark Suite. In *In Supercomputing'01* (2001), ACM Press, p. 8.

[29] VALLÉE-RAI, R., ET AL. Soot - a Java Optimization Framework. In *Proceedings of CASCON 1999* (1999), pp. 125–135.

[30] WINSKEL, G. *The Formal Semantics of Programming Languages*. MIT Press, 1993.

[31] ZHAO, J., AND SARKAR, V. Intermediate language extensions for parallelism. In *VMIL'11* (2011), pp. 333–334.