

# Parallel Data Race Detection for Task Parallel Programs with Locks

Adarsh Yoga  
Department of Computer Science  
Rutgers University  
adarsh.yoga@cs.rutgers.edu

Santosh Nagarakatte  
Department of Computer Science  
Rutgers University  
santosh.nagarakatte@cs.rutgers.edu

Aarti Gupta  
Department of Computer Science  
Princeton University  
aartig@cs.princeton.edu

## ABSTRACT

Programming with tasks is a promising approach to write performance portable parallel code. In this model, the programmer explicitly specifies tasks and the task parallel runtime employs work stealing to distribute tasks among threads. Similar to multi-threaded programs, task parallel programs can also exhibit data races. Unfortunately, prior data race detectors for task parallel programs either run the program serially or do not handle locks, and/or detect races only in the schedule observed by the analysis.

This paper proposes *Ptracer*, a parallel on-the-fly data race detector for task parallel programs that use locks. *Ptracer* detects data races not only in the observed schedule but also those that can happen in other schedules (which are permutations of the memory operations in the observed schedule) for a given input. It accomplishes the above goal by leveraging the dynamic execution graph of a task parallel execution to determine whether two accesses can happen in parallel and by maintaining constant amount of access history metadata with each distinct set of locks held for each shared memory location. To detect data races (beyond the observed schedule) in programs with branches sensitive to scheduling decisions, we propose static compiler instrumentation that records memory accesses that will be executed in the other path with simple branches. *Ptracer* has performance overheads similar to the state-of-the-art race detector for task parallel programs *SPD3*, while detecting more races in programs with locks.

## 1. INTRODUCTION

**Programming with tasks.** Task parallelism is an effective abstraction to write performance portable code. In a task parallel programming environment, the programmer specifies the tasks and the work stealing runtime distributes these tasks to the threads. A task parallel program can provide scalable speedups when the program is run on a machine with different core/thread count as the runtime dynamically balances the load between the threads. Task parallel frameworks like Cilk [17], Intel Threading Building Blocks (TBB) [39], Habanero Java [6], X10 [7], and the Java Fork-Join framework [25] have become mainstream. Given the promise of performance portable code, there are initiatives in teaching paral-

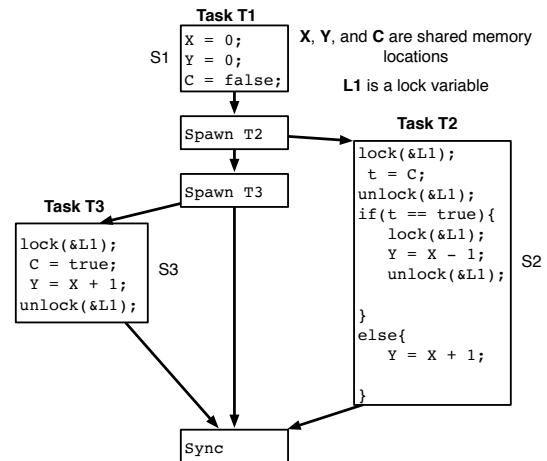


Figure 1: An example task parallel program that uses locks. There are three tasks, T1, T2, and T3. The tasks access three shared memory variables X, Y, and C. There are three regions of code without any task management constructs. They are labeled S1, S2, and S3. The program has a data race on shared memory location Y.

lelism through task parallel programming models (see Dan Grossman’s “Sophomoric Introduction to Shared Memory Parallelism and Concurrency” that advocates the use of task parallel models to teach parallelism [19]).

**Data races with task parallel programs.** A program exhibits a data race when there are multiple accesses to a shared memory location, at least one of them is a write, and there is no ordering between these accesses. In the absence of locks, a data race occurs when accesses are not ordered by task spawn/sync constructs. In the presence of locks, a data race occurs when there are two parallel accesses (one of which is a write) that are not protected by a common lock. Similar to multithreaded programs, data races in task parallel programs are usually indicators of program errors. The behavior of the program is dependent on the memory model in the presence of data races. Further, it can also cause non-deterministic execution. Figure 1 illustrates an example task parallel program with data races.

**Apparent and Feasible races.** In the terminology of Netzer and Miller [33], data races can be classified into two categories: *apparent* races and *feasible* races. Data races that appear to occur in an execution of a program primarily considering the parallel constructs but without taking the actual computation into account are

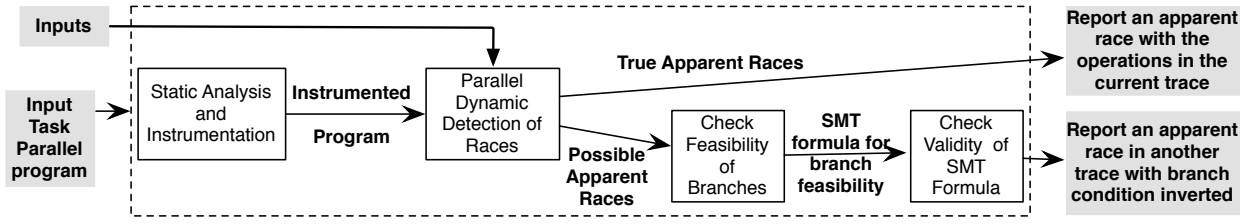


Figure 2: Workflow of our parallel data race detection algorithm for task parallel programs that use locks.

termed apparent races. Data races that occur taking into account the computation, synchronization, and parallel constructs are termed feasible races. An apparent race may not be a feasible race in some scenarios where operations in critical sections influence branches. Depending on how critical sections are scheduled, the computation itself may change when the tasks are scheduled in a different order. However, every apparent race is also a feasible race for a class of task parallel programs called Abelian programs [8], which have commutative critical sections. Identifying feasible races is a hard problem [8, 24, 33]. Detectors that aim to detect feasible races need to perform interleaving exploration, which is practically infeasible. Hence, we focus on the detection of apparent races in this paper as task parallel programs have structured parallelism.

**Classification of data race detectors.** Data race detectors can be classified into three categories based on their detection abilities: per-program, per-input, and per-schedule detectors. Per-program detectors detect possible races for all inputs and schedules for a given program. Although per-program detectors are appealing in theory, they can report a large number of false positives in practice because of approximations in the underlying static analysis [46]. Per-input detectors detect possible races in various schedules for a given input when the programs do not use locks [15, 30, 38]. Finally, per-schedule detectors detect data races in the observed schedule. They typically need to be coupled with interleaving exploration to detect races that can occur in other schedules for the same input. Further, data-race detectors can also be classified into offline, on-the-fly, or hybrid detectors depending on whether the race is detected with a postmortem analysis, during program execution, or a combination of them, respectively.

**Goals.** Our goal is to detect data races in task parallel programs with the following three objectives. First, the detector should detect races in the presence of locks because frameworks like Intel TBB [39] and Cilk [17] provide various lock implementations. Second, the detector should detect races in the observed schedule and also possible races in other schedules for a given input to either minimize or obviate the need for interleaving exploration. Third, the detector should use multiple cores available on modern processors to enable its usage with long running programs.

**Limitations of existing race detectors.** Although data race detection is a well-studied topic for multi-threaded programs, existing detectors do not satisfy our goals. FastTrack [16] is a vector-clock based detector that detects races in schedules that follow the observed happens-before ordering in multithreaded programs. When we repurpose FastTrack for tasks, FastTrack’s vector clock metadata with each shared memory location is proportional to the number of tasks, which makes it impractical for task parallel programs that create a large number of tasks. Our implementation of FastTrack aborted with out-of-memory errors with many applications (see Section 5). Prior research has also investigated numerous

techniques to detect races in task parallel programs [8, 15, 30, 37, 38]. SP-Bags [15], ESP-Bags [37], and SPD3 [38] leverage the series-parallel structure of a task parallel execution to detect races per-input for task parallel programs that do not use locks. SPD3 with support for *isolated* blocks [36] and ALL-SETS [8] detect data races per-schedule in task parallel programs. Both these approaches can detect races per-input when program has commutative critical sections (*i.e.*, Abelian programs). However, SPD3 does not support locks and ALL-SETS runs the detector serially. In this paper, we explore if it is possible to combine SPD3 and ALL-SETS to attain our objectives.

**Our approach.** Inspired by SPD3 [38] and ALL-SETS [8], we propose an on-the-fly dynamic data race detector, PTRacer, for task parallel programs with locks. PTRacer uses the dynamic program structure tree (DPST) representation from SPD3 to determine whether two accesses can happen in parallel. It borrows the idea of tracking the set of locks held before an access with each entry in the metadata space from ALL-SETS. The key challenge is in maintaining appropriate metadata when there are multiple readers and writers to a given shared memory location to enable effective and efficient detection of races.

Further, branch statements in a task parallel program can be influenced by scheduling decisions when the program uses locks. Such branches are called schedule-sensitive branches (SSB)[22]. In the presence of SSBs, the dynamic trace observed by the analysis will likely not have memory operations from all schedules (*e.g.*, from the path not-taken at the branch statement). We explore if it is possible to detect races that happen in other schedules in the presence of SSBs.

To accomplish our objectives, we designed PTRacer with three components: (a) static analysis and instrumentation component to instrument shared memory accesses and identify accesses in the other path with simple branch statements, (2) a parallel dynamic race detection component that extends SPD3 to handle locks, and (3) a diagnosis component that classifies some of the races reported by the parallel dynamic analysis as infeasible if the branch is not schedule sensitive. Figure 2 provides an overview of the various components in PTRacer.

**Static analyses and instrumentation to identify shared memory accesses.** This component of PTRacer has two goals: instrument shared memory accesses and identify accesses that can be executed in the other path at a branch statement. PTRacer uses a compiler pass to instrument the program with calls to a dynamic race detection library. The compiler pass identifies branch statements and records the memory accesses executed in the other path at a simple branch statement with calls to the race detection library. When the branching structure involves loops or has nested branches, the compiler pass informs the user that the dynamic race detection will be restricted to the detection of apparent races in the

observed schedule and schedules that are permutations of memory operations in the observed schedule.

**Parallel dynamic analysis to detect races.** The dynamic data race detection component of `Ptracer` constructs the DPST as the program executes and performs dynamic data race detection when the race detection library calls execute. Hence, the data detection happens in parallel. `Ptracer` maintains two reads and two writes with each distinct set of locks held before an access to a shared memory location in the metadata space for each shared memory location. Maintaining information about two reads and a single write is sufficient when accesses are performed without holding any locks.

When there are multiple writes to a shared memory location with same set of locks held, `Ptracer` maintains two write operations so that all other parallel write operations to the same location are in the subtree under the least common ancestor (LCA) of the two writes in the DPST. Any future write or a read operation that may execute in parallel with any of the not-maintained accesses will execute in parallel with at least one of the two writes maintained in the metadata space. The dynamic data race detector in `Ptracer` reports a race when two accesses are not protected by a common lock.

The races reported by the dynamic race detector are either true apparent races that involve operations in the observed schedule or possible apparent races that involve races between operations from the observed schedule and operations in the not-taken path at a branch statement.

**Diagnosis phase.** Finally, the diagnosis component of `Ptracer` filters out possible races by checking the schedule sensitivity of the branch resulting in the race. The diagnosis component runs the program again to obtain a detailed trace. It encodes the detailed trace and the inverted branch condition of the schedule sensitive branch involved with the possible apparent race as a first-order logic formula and checks its satisfiability (similar to prior work [22, 23, 27]). If the formula is satisfiable, then the diagnosis component reports the race as an apparent race to the user.

**Guarantees.** `Ptracer` detects apparent races for a given input for Abelian programs similar to `ALL-SETS`. When there are multiple races involving a shared memory location, `Ptracer` reports a single race to the user. `Ptracer` detects apparent races in the observed schedule and in schedules that are permutation of the memory operations in the observed schedule for non-Abelian programs. Some of these races may not be feasible if the computation in the program forbids them. Even the infeasible apparent races can suggest likely program errors as task parallel programs typically have structured communication. The race detection involving operations from non-taken paths is best-effort (*i.e.*, it can miss races with SSBs) due to the limitations of our static analysis.

Our prototype detector `Ptracer` detects data races in Intel TBB programs that use locks. The prototype detects all races in our test suite, which has unit tests with locks, without false positives and without requiring interleaving exploration. In contrast, `FastTrack` misses many races and `SPD3` reports false positives as it does not handle locks. `Ptracer` is usable with long-running applications and has performance overhead similar to `SPD3`.

## 2. BACKGROUND

This section provides background on Dynamic Program Structure Tree (DPST) representation of a task parallel execution to identify parallel accesses. As we use DPST and build on `SPD3` [38], we also provide a brief background on the `SPD3` data race detector for task parallel programs.

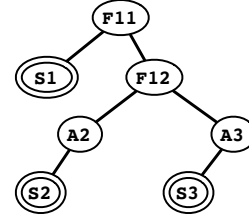


Figure 3: DPST for the sample task parallel program in Figure 1 after it has executed all statements. There are three step nodes ( $S_1$ ,  $S_2$ , and  $S_3$ ) in the DPST, which are depicted by nodes with two circles and are executed by tasks  $T_1$ ,  $T_2$ , and  $T_3$  respectively. The step nodes  $S_2$  and  $S_3$  can occur in parallel because  $LCA(S_2, S_3)$  is  $F_{12}$  and the left child of  $F_{12}$  is an **async** node and it is an ancestor of  $S_2$ . The step nodes  $S_1$  and  $S_2$  cannot occur in parallel as  $LCA(S_1, S_2)$  is a finish node  $F_{11}$  and its left child that is also an ancestor of  $S_1$  is not an async node.

### 2.1 Dynamic Program Structure Tree

The execution of a task parallel program results in a series-parallel execution graph. The series-parallel execution graph can be used to determine whether two accesses by different tasks can logically execute in parallel [8, 15, 30]. The graph can be used to find races in other schedules not encountered in the current execution trace when these programs do not use locks. However, the program has to be executed serially, which results in performance overheads.

To address the problem of serial execution with data race detection, Raman *et al.* [38] proposed an approach to check whether two accesses can logically execute in parallel using an ordered tree called the Dynamic Program Structure Tree (DPST). DPST captures the dynamic parent-child relationship between tasks, which enables parallel race detection with `SPD3` [38].

The DPST consists of three types of nodes: (a) **step** nodes, (b) **finish** nodes, and (c) **async** nodes. A step node in the DPST represents the maximal sequence of instructions without any task spawn (for task creation) and sync (join) statements. All computation and memory accesses occur in the step nodes. Hence, every memory access has a corresponding step node associated with it. Further, the step nodes are always leaf nodes in the DPST.

The async nodes capture the spawning of a task by a parent task. The descendants of an async node execute asynchronously with the remainder of the parent task. A finish node is created when a task spawns a child task and waits for the child (and its descendants) to complete. A finish node is the parent of all async, finish and step nodes directly executed by its children or their descendants. The siblings of a particular node in a DPST are ordered left-to-right to reflect the left-to-right sequencing of computation of their parent task.

DPST’s construction ensures that all internal nodes are either async or finish nodes. A path from a node to the root and the left-to-right ordering of siblings in a DPST does not change even when nodes are added to the DPST during execution. The construction of the DPST ensures that two step nodes  $S_1$  and  $S_2$  (assuming  $S_1$  is to the left of  $S_2$ ) can execute in parallel if the least common ancestor of  $S_1$  and  $S_2$  (*i.e.*,  $LCA(S_1, S_2)$ ) in the DPST has an immediate child  $A$  that is an async node and is also an ancestor of  $S_1$ .

Consider the example task parallel program in Figure 1 with three tasks  $T_1$ ,  $T_2$ , and  $T_3$ . Figure 3 presents the DPST after all tasks and instructions in the program in Figure 1 have executed. There are three step nodes:  $S_1$ ,  $S_2$ , and  $S_3$ . There are two finish nodes:  $F_{11}$  that corresponds to the implicit finish with the main

task and F12 that corresponds to the collection of tasks T1 and T2 followed by a sync statement. The step nodes S2 and S3 can occur in parallel since the LCA(S2, S3) is F12 and its left child is an async node. In contrast, step nodes S1 and S2 cannot occur in parallel since the LCA(S1, S2) is F11 and its left child that is also an ancestor of S1 is S1, which is not an async node. Similarly, step nodes S1 and S3 cannot occur in parallel.

## 2.2 SPD3 Race Detector

Any dynamic race detector needs to determine if two accesses (at least one of them is a write) can happen in parallel and track accesses to the same location. SPD3 [36, 38] uses the DPST to determine if two accesses can occur in parallel. It maintains shadow memory for each memory location that tracks tasks that have accessed the same location.

Rather than maintaining information about every access to a shared memory location by tasks in shadow memory, SPD3 maintains a total of two reads ( $r_1$  and  $r_2$ ) and a write ( $w_1$ ) with every shared memory location. It is sufficient to maintain information about one write as all other writes should either occur in series or constitute a data race. However, there can be multiple readers and it is necessary to maintain information about them. When there are multiple parallel readers to a location, SPD3 stores two reads  $r_1$  and  $r_2$  such that the subtree under  $LCA(r_1, r_2)$  in the DPST includes other reads. In contrast to vector clock based detectors, SPD3 maintains constant number of access history entries with each monitored shared memory location irrespective of the number of tasks.

SPD3 detects races for a given input by examining a single trace provided the task parallel program does not use locks. Next, we describe how to handle locks and propose a technique to detect races not only in the current trace but also in other schedules for a given input.

## 3. APPROACH

Our goal is to design an on-the-fly data race detector for task parallel programs with the following attributes: (1) runs in parallel, (2) handles programs that use locks, and (3) detects data races that occur in different schedules for a given input by examining a single trace. We do not need to store long traces with an on-the-fly detector. A parallel data race detector reduces performance overheads by leveraging multi-cores. Handling locks enables us to detect races in applications written with frameworks such as Intel TBB [39] and Cilk [17] that support locks. Detecting races that can happen in other schedules by examining a single schedule for a given input either minimizes or obviates (in the best case) the need for interleaving exploration.

We are primarily focused on detecting *apparent* races, which are races that appear to occur taking into account the parallel constructs in the program [33]. In presence of critical sections, some of these apparent races may not be feasible when the actual computation performed by the task is considered along with the parallel and synchronization constructs. Detecting feasible races typically requires interleaving exploration and covering all interleavings is not possible in practice. However, every apparent race is a feasible race for a class of programs (*i.e.*, Abelian programs) with commutative critical sections [8].

When the program contains non-commutative critical sections, we investigate if it possible to detect races that can happen in other schedules to minimize the need for interleaving exploration. In such scenarios, our goal is to detect apparent races that can happen in other schedules, which perform the same shared memory accesses but possibly in a different order, for a given input. This

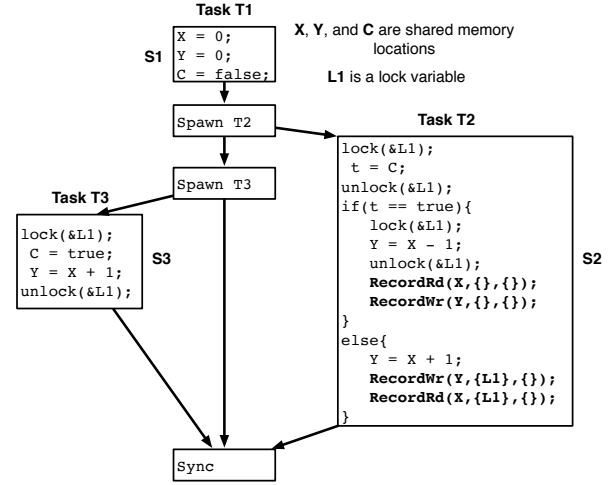


Figure 4: The task parallel program in Figure 1 instrumented with RecordRd and RecordWr instrumentation calls using static analysis. There is a RecordWr(Y, {}, {}) in the if-block that is executed when C is true in Task T2 because shared memory location Y is written in the else-branch without any lock acquisitions and releases from the start of the else-block. Similarly, there is RecordWr(Y, {L1}, {}) in the else-block because memory location Y is written in the if-branch with lock L1 acquired and no locks have been released from the beginning of the if-block.

guarantee is similar in spirit to the guarantees aimed by predictive testing techniques for multi-threaded programs [21, 44, 45, 47].

In the presence of critical sections, the branch statements in the program can also be influenced by the scheduling of critical sections. Such branches are called *schedule sensitive branches* (SSBs). A schedule observed by the dynamic analysis may not contain memory accesses from the not-taken path of a schedule sensitive branch. We propose a static instrumentation technique to record the memory accesses that will be executed in the not-taken branch. This approach enables us to detect apparent races that can occur in different schedules of a program with SSBs for the same input without requiring interleaving exploration. To filter false positives when the branch is not schedule sensitive, we encode the trace and the inverted branch condition as a first-order logic formula and check its satisfiability.

**Three components of our proposed detector.** Our proposed detector PTRacer consists of three components to accomplish the above goals: (1) static analysis and instrumentation component to instrument the program with calls to the dynamic race detection library both to construct the DPST, record the shared memory accesses in the presence of branches, and to detect races, (2) a parallel dynamic analysis component that executes when the task parallel program executes and detects races, and (3) a diagnosis component that checks the feasibility of the reported races involving memory accesses from the not-taken path at a branch statement. Figure 2 illustrates the three components of PTRacer, which we describe in detail below.

### 3.1 Static Instrumentation Component

The static analysis and instrumentation component has three objectives: (1) add instrumentation to identify task management constructs to build the DPST at runtime, (2) add instrumentation to identify shared memory accesses and lock operations to perform dynamic race detection, and (3) add instrumentation to identify



```

procedure DATARACEDETECTOR( $l, S, A, LS$ )
   $AH \leftarrow Metadata(l)$ 
  for all  $p \in AH$  do
    if  $p.LS \cap LS = \emptyset$  then
      if  $A = Rd \wedge DMHP(S, p.W_1)$  then
        Report write-read race between  $p.W_1$  and  $S$ 
      end if
      if  $A = Rd \wedge DMHP(S, p.W_2)$  then
        Report write-read race between  $p.W_2$  and  $S$ 
      end if
      if  $A = Wr \wedge DMHP(S, p.W_1)$  then
        Report write-write race between  $p.W_1$  and  $S$ 
      end if
      if  $A = Wr \wedge DMHP(S, p.W_2)$  then
        Report write-write race between  $p.W_2$  and  $S$ 
      end if
      if  $A = Wr \wedge DMHP(S, p.R_1)$  then
        Report read-write race between  $p.R_1$  and  $S$ 
      end if
      if  $A = Wr \wedge DMHP(S, p.R_2)$  then
        Report read-write race between  $p.R_2$  and  $S$ 
      end if
    end if
    if  $p.LS = LS$  then  $\triangleright$  Update the metadata for lockset  $LS$ 
      if  $A = Wr$  then
        if  $\neg DMHP(S, p.W_1) \wedge \neg DMHP(S, p.W_2)$  then
           $p.W_1 \leftarrow S$ 
           $p.W_2 \leftarrow null$ 
        end if
        if  $DMHP(S, p.W_1) \wedge DMHP(S, p.W_2)$  then
           $lca_{12} \leftarrow LCA(p.W_1, p.W_2)$ 
           $lca_{1s} \leftarrow LCA(p.W_1, S)$ 
           $lca_{2s} \leftarrow LCA(p.W_2, S)$ 
          if  $lca_{1s} >_{dpst} lca_{12} \vee lca_{2s} >_{dpst} lca_{12}$  then
             $p.W_1 \leftarrow S$ 
          end if
        end if
      end if
      if  $A = Rd$  then
        if  $\neg DMHP(S, p.R_1) \wedge \neg DMHP(S, p.R_2)$  then
           $p.R_1 \leftarrow S$ 
           $p.R_2 \leftarrow null$ 
        end if
        if  $DMHP(S, p.R_1) \wedge DMHP(S, p.R_2)$  then
           $lca_{12} \leftarrow LCA(p.R_1, p.R_2)$ 
           $lca_{1s} \leftarrow LCA(p.R_1, S)$ 
           $lca_{2s} \leftarrow LCA(p.R_2, S)$ 
          if  $lca_{1s} >_{dpst} lca_{12} \vee lca_{2s} >_{dpst} lca_{12}$  then
             $p.R_1 \leftarrow S$ 
          end if
        end if
      end if
    end if
  end for
  if  $LS \notin Metadata(l)$  then
    Create a new lockset  $LS$  and add a new entry for  $l$ 
  end if
  return
end procedure

```

Figure 5: Algorithm to check for a data race on memory access to location  $l$  by the step node  $S$  with access type  $A$  and lockset  $LS$ .  $Metadata(l)$  function returns the access history in shadow memory associated with location  $l$ . The predicate  $DMHP$  (Dynamic May Happen in Parallel) is used to determine if two accesses are parallel. The predicate  $DMHP(S_i, S_j)$  returns true if the step nodes  $S_i$  and  $S_j$  in the DPST may execute in parallel.  $LCA(S_i, S_j)$  returns the least common ancestor node of  $S_i$  and  $S_j$ .

shared memory operations that are possibly executed in the other path in the presence of a schedule-sensitive branch statement, which is inspired by a similar attempt for property driven pruning with dy-

namic partial order reduction [48].

The static analysis and instrumentation phase can be performed either on the source code or within the compiler. We perform this phase within the compiler for mainly two reasons: (a) the compiler already performs various analyses to identify thread local accesses and the branch statements, and (b) the instrumentation can be performed on optimized code, which reduces the performance overhead of race detection.

The compiler pass that adds calls to the race detection library to identify task management constructs and shared memory accesses is straightforward, which we omit for space constraints. The interesting aspect is the addition of instrumentation to identify memory accesses performed in both the taken and the not-taken paths in the presence of schedule sensitive branches. Our algorithm to record memory accesses is described below. First, the compiler pass identifies conditional and unconditional branch statements and their corresponding join statements. Second, the compiler pass identifies the set of memory operations performed, the set of locks acquired and released before every memory operation in the taken and the not-taken branch from the beginning of the branch for each branch statement and its corresponding join statement. If the branch condition does not dominate either the memory access or the lock variable, then we need to perform additional work to make these accesses visible in the other branch. In such scenarios, we add the backward static program slice of the memory access and/or the lock variable in the other branch. Finally, the compiler inserts calls to the runtime library to record the memory operation performed in the other path. If a memory access  $A$  is written in the else-block of a schedule sensitive branch with  $L_a$  locks acquired and  $L_r$  locks released from the beginning of the else-block till memory operation  $A$ , then the compiler introduces a runtime call  $RecordWr(A, L_a, L_r)$  in the if-block.

Figure 4 shows the additional  $RecordWr$  and  $RecordRd$  in the if-branch and the else-branch. In the if-branch, additional  $RecordWr$  and  $RecordRd$  instrumentation correspond to memory accesses in the else-branch. The primary effect of this instrumentation is that we will detect data races that would occur with the operations in the else-branch even when the trace observed during the dynamic analysis contains only the operations from the if-branch.

**Limitations.** We primarily focus on simple branch statements that are non-nested and not part of loops to record memory accesses from the not-taken path. Our static analysis informs the user about the presence of non-nested branches and branches that are part of loops. In the presence of such branches, our framework still detects apparent races that occur in other schedules whose memory operations are a permutation of the memory operations in the observed trace. The taken and not-taken paths at a branch can include function calls provided it is non-recursive without nested-branches and loops. We chose this design point to avoid false apparent races. However, our race detector will miss some races given these limitations.

### 3.2 Parallel Dynamic Data Race Detector

$PTracer$  detects data races when the program executes race detection library calls introduced by the static instrumentation component. As tasks execute in parallel, the race detection also happens in parallel. The dynamic race detector component of  $PTracer$  maintains two pieces of information at runtime. DPST is constructed at runtime to determine if two accesses can occur in parallel. The metadata is maintained with each shared memory location that provides information about prior accesses by various tasks.

**Metadata design.** A naive approach to detect apparent races (in the observed schedule and other schedules involving the same memory operations) would maintain a list of accesses performed by various tasks with each shared memory location. As each access to a shared memory location can occur with different sets of locks held, the access history should also maintain information about the set of locks held (lockset) before performing a memory access. However, such an approach would make the metadata proportional to the number of dynamic memory accesses and is infeasible in practice.

Our contribution is in designing a dynamic data race detection algorithm that maintains a constant number of access history entries, which is independent of the number of tasks and the number of dynamic memory accesses, while handling locks. Our metadata for each shared memory location contains four access history entries (step nodes of two reads:  $R_1$  and  $R_2$ , and two writes:  $W_1$  and  $W_2$ ) for each distinct set of locks held before the access. Although the size of the metadata is proportional to the number of distinct sets of locks held for each memory location, we observe in practice that each shared memory location is accessed with similar sets of locks. In summary, the access history with each shared memory location can be conceptually viewed as an array of data nodes, where each data node contains the unique lockset and step nodes corresponding to two reads ( $R_1$  and  $R_2$ ) and two writes ( $W_1$  and  $W_2$ ).

**Metadata checks on a shared memory access.** Figure 5 provides the algorithm for checking the metadata on a shared memory access. The algorithm iterates over all access history entries corresponding to each lockset in the metadata space. First, the algorithm checks if the intersection of the lockset of the current access and each lockset in the metadata space is empty. If the intersection is empty, two accesses have been performed without a common lock and PTRacer reports a race if two accesses can occur in parallel and at least one of them is a write.

**Updating the read metadata.** After the check, the algorithm in Figure 5 updates the metadata corresponding to the appropriate lockset. If the current access is a read access, then the metadata is updated similar to SPD3 [38] except that the access history entries for a particular lockset are updated. If the current read access is in series with both the reads ( $R_1$  and  $R_2$ ) corresponding to the current lockset, then  $R_1$  is set to the current access and  $R_2$  is set to null (a unique empty value). When there are multiple readers that can execute in parallel (*i.e.*, two existing readers  $R_1$  and  $R_2$  in the metadata space and current access), PTRacer maintains two reads in the metadata space such that the subtree under  $LCA(R_1, R_2)$  includes all reads similar to SPD3. The key insight is that any future access that can have a data race with the not-stored reads will also have a data race with  $R_1$  and/or  $R_2$ .

**Updating the write metadata for a lockset.** Updating the metadata in the presence of write operations and locksets requires some thought in comparison to SPD3. When writes are performed without locks, any two parallel writes is a data race. When tasks use locks, two writes can happen in parallel but may be protected by the same lock. Hence, they do not constitute a data race. When PTRacer sees multiple parallel writes (current access,  $W_1$  and  $W_2$  in the metadata space) with the same lockset, it needs to identify two writes to maintain in the metadata space. Similar to multiple parallel reads, PTRacer maintains two writes in the metadata space such that the subtree under  $LCA(W_1, W_2)$  includes all writes. Any future access that can race with one of the not-stored writes will also race with at least one of  $W_1$  or  $W_2$ . Maintaining only two reads and two writes with each distinct set of locks

Time	Observed trace	Metadata for Y
1.	T1/S1: X = 0;	[[]]
2.	T1/S1: Y = 0;	[[{}], S1, null, null, null]]
3.	T1/S1: C = false;	[[{}], S1, null, null, null]]
4.	T1 : Spawn T2;	[[{}], S1, null, null, null]]
5.	T1 : Spawn T3;	[[{}], S1, null, null, null]]
6.	T3/S3: lock(&L1);	[[{}], S1, null, null, null]]
7.	T3/S3: C = true;	[[{}], S1, null, null, null]]
8.	T3/S3: Y = X + 1;	[[{}], S1, null, null, null], [[L1], S3, null, null, null]]
9.	T3/S3: unlock(&L1);	[[{}], S1, null, null, null], [[L1], S3, null, null, null]]
10.	T2/S2: lock(&L1);	[[{}], S1, null, null, null], [[L1], S3, null, null, null]]
11.	T2/S2: t = C;	[[{}], S1, null, null, null], [[L1], S3, null, null, null]]
12.	T2/S2: unlock(&L1);	[[{}], S1, null, null, null], [[L1], S3, null, null, null]]
13.	T2/S2: if(t == true)	[[{}], S1, null, null, null], [[L1], S3, null, null, null]]
14.	T2/S2: lock(&L1);	[[{}], S1, null, null, null], [[L1], S3, null, null, null]]
15.	T2/S2: Y = X - 1;	[[{}], S1, null, null, null], [[L1], S3, S2, null, null]]
		-> Two writes to Y with lock set L1 by step nodes S3 and S2.
16.	T2/S2: unlock(&L1);	[[{}], S1, null, null, null], [[L1], S3, S2, null, null]]
17.	T2/S2: RecordRd(X, {}, {})	[[{}], S1, null, null, null], [[L1], S3, S2, null, null]]
18.	T2/S2: RecordWr(Y, {}, {})	[[{}], S2, null, null, null], [[L1], S3, S2, null, null]]
		-> Write-Write race for Y based on its access in the other branch as S2 and S3 occur in parallel
19.	T1: Sync	

Figure 6: Illustration of race detection with a concrete trace of the program in Figure 4. The observed trace provides the instruction executed, the task, and the step node performing the operation observed in the trace. The metadata in shadow memory for the shared memory variable Y is also shown. The metadata for a shared memory location is a list of access histories with each lockset. There are four access histories for each lockset ( $W_1$ ,  $W_2$ ,  $R_1$ ,  $R_2$ ). On time step 2, Y is written by step node S1 in Task T1 without holding any lock. We create a new entry for the empty lockset and update the access history corresponding to the write by S1 *i.e.*,  $[[{}], S1, null, null, null]$ . Similarly when Y is written with lock set L1 by step node S3 in task T3 at time step 8, the race detector checks if this access results in a race according to existing history and updates the metadata of Y with a new lock set L1 and the current write. It is important to note that when Y is written by step node S2 in task T2 at time step 15, there is already a write to Y in the metadata space with lock set L1. Since this write by S2 can occur in parallel with the existing write, we maintain both writes in the metadata for Y. Finally, the record instrumentation enables us to find races that would occur when the not-taken branch is executed on a different schedule.

enables PTRacer to detect apparent races both in the observed schedule and other schedules for the same input. Figure 6 illustrates the metadata for shared memory variables, checks, and the metadata update actions performed after each statement in the observed trace.

### 3.3 Diagnosis Phase

The parallel data race detection algorithm described above reports two kinds of apparent races, which we call true races and possible races. A true race is a data race that occurs between the operations of the observed trace without involving memory operations from the RecordWr and RecordRd instrumentation. A possible race is a data race that involves shared memory operations from the other-branch instrumentation. If the branch is not schedule sensitive, then these possible races will never manifest for a given input. Hence, we propose a diagnosis phase to identify whether the branch is schedule sensitive when our parallel data race algorithm reports a possible race. We divide the diagnosis phase into two components: execution trace generator that generates per-task execution traces for the reported possible race, and a constraint generator that checks if the branch statement responsible for the reported possible race is schedule sensitive by transforming the trace into a first-order logic formula and checking its satisfiability using an SMT solver. Our constraint generation and checks for schedule

a) Trace Representation		b) Constraints
<b>Per-task trace</b>	<b>Canonicalized trace</b>	<b>Per-task Order Constraints:</b>
1. X = 0;	WX@1.1 = 0;	$(W_{1,1} < W_{1,2} < W_{1,3})$
2. Y = 0;	WY@1.2 = 0;	$\wedge (L_{2,1} < R_{2,2} < W_{2,2} < U_{2,3} < R_{2,4}$
3. C = false;	WC@1.3 = false;	$< L_{2,5} < R_{2,6} < W_{2,6} < U_{2,7})$
4. Spawn T2;	SpawnT2@1.4	$\wedge (L_{3,1} < W_{3,2} < R_{3,3} < W_{3,3} < U_{3,4})$
5. Spawn T3;	SpawnT3@1.5	
6. Sync;	SyncT2,T3@1.6	
<b>Per-task trace</b>	<b>Canonicalized trace</b>	<b>Synchronization Constraints:</b>
1. lock(L1);	LL1@2.1;	$(U_{3,4} < L_{2,1}) \vee (U_{2,3} < L_{3,1})$
2. t = C;	RC@2.2;	
	Wt@2.2 = C@2.2;	<b>Read-Write Constraints:</b>
3. unlock(L1);	UL1@2.3;	$((C_{2,2} = \text{false}) \wedge (W_{1,3} < R_{2,2}) \wedge$
4. if (t == true) {	Rt@2.4	$(W_{3,2} < W_{1,2} \vee R_{2,2} < W_{3,2}))$
5. lock(L2);	LL2@2.5;	$\vee ((C_{2,2} = \text{true}) \wedge (W_{3,2} < R_{2,2}) \wedge$
6. Y = X - 1;	RX@2.6;	$(W_{1,3} < W_{3,2} \vee R_{2,2} < W_{1,3}))$
	WY@2.6 = X@2.6 - 1;	$\wedge ((t_{2,4} = C_{2,2}) \wedge (W_{2,2} < R_{2,4}))$
7. unlock(L2);	UL2@2.7;	
<b>Negated Branch Check:</b> t@2.4 != true		<b>Spawn-Sync Constraints:</b>
<b>Per-task trace</b>	<b>Canonicalized trace</b>	$(W_{1,3} < L_{2,1}) \wedge (W_{1,3} < L_{3,1})$
1. lock(L1);	LL1@3.1;	
2. C = true;	WC@3.2 = true;	<b>Negated Branch Check Constraint:</b>
3. Y = X + 1;	RX@3.3;	$t_{2,4} != \text{true}$
	WY@3.3 = X@3.3 + 1;	
4. unlock(L1);	UL1@3.4	

Figure 7: (a) Concrete per-task traces and canonicalized trace representation. For every statement in the per-task trace, we create an order variable. The order variable for a write operation performed by statement  $j$  in task  $T_i$  is represented by  $W_{i,j}$ . Similarly, the order variables for the read, lock, and unlock operations performed in statement  $j$  by task  $T_i$  is represented by  $R_{i,j}$ ,  $L_{i,j}$ , and  $U_{i,j}$  respectively. The elements within the same task are ordered. Hence, we have  $W_{1,1} < W_{1,2} < W_{1,3}$ . There are two value variables ( $C_{2,2}$  and  $t_{2,4}$ ) representing symbolic read operations that directly or indirectly influence the branch condition. The read-write constraints connect the value variables and the order variables. The synchronization and spawn-sync constraints further restrict the feasible orderings. If the negated branch condition is satisfiable with the constraints, then the branch is a schedule sensitive branch and the race is reported to the user.

sensitive branches is inspired by TAME [22] and DSP [27]. We enforce an ordering corresponding to the possible race and repurpose prior approaches to a task-based context.

**Per-task execution trace generator.** When the dynamic race detector reports a possible race, we enforce the schedule corresponding to the possible race. This step is necessary because the dynamic race detection algorithm does not log the trace as it wants to detect races with a low performance overhead. We also generate canonicalized per-task execution traces. The canonicalized per-task execution trace captures all loads/stores, synchronization constructs, and the branch condition corresponding to the possible race. To check if the branch is schedule-sensitive, we need to check if the negation of the branch condition involved in a possible race is satisfiable. Figure 7(a) shows the per-task trace obtained for an execution of the program in Figure 4 and its canonicalized trace.

**Constraint generator.** Using the per-task traces, we construct a first-order logic formula to check for the satisfiability of the negated branch condition. Inspired by DSP [27], we create two types of variables: *order variables* and *value variables*. The *order variables* are used to encode the position of the operation in the per-task trace. The *value variables* are used to symbolically encode the read operations. Since, we are specifically interested in checking the satisfiability of the negated branch condition, we will create value variables for only those reads that directly or indirectly influence the branch condition. We generate a formula  $\Phi$  that relates these order and value variables using the task-parallel execution constraints and checks the negated branch condition.

$$\Phi = \phi_{po} \wedge \phi_{sync} \wedge \phi_{rw} \wedge \phi_{ss} \wedge \phi_{br}$$

where  $\phi_{po}$  is the constraint encoding the order of execution of operation in a given task,  $\phi_{sync}$  is the constraint encoding the possible orders of execution among the synchronization statements among tasks,  $\phi_{rw}$  is the constraint that encodes the data flow between various accesses to the same location,  $\phi_{ss}$  is the constraint encoding the parent-child relationship between tasks, and  $\phi_{br}$  is the constraint representing the negated branch condition. Satisfiability of  $\Phi$  implies that the branch involved in a possible race is a schedule sensitive branch and the possible races reported are indeed feasible.

**Generating the constraints.** The per-task order constraints ( $\phi_{po}$ ) ensure that the operations within the same task are ordered. The synchronization constraints ( $\phi_{sync}$ ) order the lock and unlock operations performed by the tasks on the same lock. Let  $L_{i,m}/U_{i,n}$  and  $L_{j,p}/U_{j,q}$  be two lock/unlock order variables from two different tasks ( $i \neq j$ ) on the same lock. There are two possible orderings in this scenario. When the lock statement from task  $i$  is executed first, then  $L_{j,p}$  is executed only after the unlock statement  $U_{i,n}$ , which results in an ordering constraint  $U_{i,n} < L_{j,p}$ . Otherwise, the lock statement from task  $j$  is executed first, which results in an ordering constraint  $U_{j,q} < L_{i,m}$ . Hence, the synchronization constraint is a disjunction of these two constraints.

The read-write constraint ( $\phi_{rw}$ ) connects the value variable and order variables corresponding to writes to the same location. For every read operation  $r$ , our constraint specifies that it reads the value of a particular write  $w$  if  $w$  happens before  $r$  and every other write happens before  $w$  or after  $r$ . Since we check for the satisfiability of the negated branch condition, we consider only those reads and writes that directly or indirectly affect the branch condition. Figure 7(b) illustrates the constraints generated for the per-task traces in Figure 7(a).

In summary, if the constraints generated are satisfiable, then the branch is a schedule sensitive branch and the not-taken path at a branch statement will be executed for the same input. Hence, we report all such possible races involved with schedule sensitive branches to the user.

## 4. IMPLEMENTATION

This section describes the metadata encoding and the implementation optimizations that we use to reduce the performance overhead of data race detection.

### 4.1 Metadata Organization

The metadata for each shared memory location is stored in shadow memory. We implement shadow memory using a two-level lookup trie data structure as it provides the ability to shadow every address in memory efficiently. A trie is a page-table like structure where each level is accessed using few bits from the address whose metadata is being looked up. PTRacer maps a 48-bit virtual address space using a two-level trie [31]. The first-level trie mappings are allocated at program initialization and the second level entries are allocated on demand when a memory location is touched for the first time, which reduces the memory overhead.

**Metadata encoding.** The metadata associated with a shared memory location is a list of four access history entries for each lockset. Maintaining linked data structures in the shadow space increases the performance overhead. In practice, we observe that most shared memory locations are accessed with a small number of locksets. Hence, we accelerate the common case (*i.e.*, accesses with few locksets) by organizing the entry in shadow memory as a constant-sized array of data nodes. Each data node represents the access history for a given lockset. If a shared memory location is accessed with more locksets than the constant-sized array of

data nodes, then we resort back to the slow list representation for that shared memory location. Each data node contains five 64-bit values: a 64-bit value encoding the lockset, two 64-bit values for representing the step nodes performing the reads:  $R_1$  and  $R_2$ , and two 64-bit values for representing the step nodes performing the writes:  $W_1$  and  $W_2$ . These implementation techniques enabled us to successfully run parallel data race detection on large and long running programs.

## 4.2 Optimizations

We observed three major opportunities for reducing the performance overhead in our implementation: (1) choosing appropriate data structures for the DPST, (2) identifying redundant checks, and (3) identifying redundant LCA queries on the DPST. We describe these optimizations below.

**Overlay DPST in a linear array.** Rather than building the DPST using a linked n-ary tree data structure, we optimize the layout of the DPST by overlaying the tree in a linear array of nodes. We maintain parent-child relationship in such an overlay by maintaining the index of the parent node with each child node. We achieve better locality, avoid pointer chasing code, and avoid the cost of frequent dynamic allocations by overlaying the DPST in a linear array of nodes. Further, this representation also reduced the overhead of a single LCA query when compared to the linked data structure because it eliminated several pointer indirections in the traversal of the DPST.

**Access caching.** We observed that there were multiple accesses to the same location with the same lockset from a given step node. However, we were not able to prove that they are redundant accesses through static analysis. We observe that when there are multiple accesses of the same type with the same lockset in a step node, then it is sufficient to perform the check once and store the metadata for only one access. We reduce the overhead of metadata checking and propagation by caching accesses performed in the task, and not performing the check on accesses that have an entry in the cache with the same lockset and access type.

**LCA caching.** LCA queries are expensive even after optimizing the layout of the DPST because each query can traverse a large number of nodes. Moreover, our data race detection algorithm performs LCA queries on each access to check and propagate metadata. We observe that even when previously unseen addresses are being checked, there are opportunities to cache LCA queries. LCA queries check whether two step nodes can occur in parallel. When these step nodes have been previously accessed in a LCA query, it is not necessary to perform it again as the series-parallel relationship between the tasks does not change as nodes are being added to the graph. Hence, we cache the frequently performed LCA queries to reduce overhead of the repeated traversals of the DPST. These optimizations not only reduced the overhead of our implementation but also reduce the overhead of SPD3, the baseline that we compare against in our evaluation.

## 5. EXPERIMENTAL EVALUATION

This section describes our prototype, implementation optimizations, benchmarks, and experimental evaluation to measure effectiveness in race detection and the performance overhead of data race detection.

**Prototype.** Our prototype `Ptracer` is designed for C++ programs that use Intel Threading Building Blocks (TBB) for task parallelism. It includes a compiler intermediate representation instrumenter, a race detection library that performs runtime race detec-

Benchmark	No. of accesses	No. of other branch accesses	No. of LCAs	Percent. of unique LCAs	Percent. of access cache hits
blackscholes	140M	0	253M	67.27	49.05
bodytrack	32.48M	0	96.97M	24.08	0.8
fluidanimate	27.49M	0	74.08M	41.21	86.77
streamcluster	257M	63,742	854M	60.93	70.86
swaptions	301M	56,200	924M	63.11	64.45
convexhull	30.07M	26,386	19.11M	61.37	99.82
delrefine	153M	0	328M	53.43	0
deltriang	20M	366	64.36M	32.87	0
karatsuba	115M	22,438	152M	53.32	71.96
kmeans	118M	582	147M	30.19	54.57
nearestneigh	76M	0	134M	64.64	48.72
raycast	128M	11,704	655M	74.85	1.03
sort	11.74M	3768	4.14M	45.78	97.87

Table 1: We report the number of dynamic shared memory accesses, the number memory accesses due to record instrumentation to capture operations from the not-taken path at a branch statement, the number of least common ancestor queries, the percentage of unique LCA queries, and the percentage of accesses that hit in the access cache for each benchmark. We use M for million in the table.

tion, an execution trace generator, and a constraint generator. The instrumenter is implemented as a compiler pass in Clang+LLVM-3.7. It inserts calls to the race detection library at shared memory accesses, synchronization statements, and task management statements. We use the demangled name of the library calls to identify synchronization and task management statements. The instrumenter also identifies accesses performed in both the taken and not-taken paths of a branch statement.

The race detection library is written in C++. The runtime library builds the DPST, performs metadata propagation and checks for data races as described in Figure 5. The constraint generator is written in Python. It parses the per-task traces, creates order and value variables, and constructs the first-order logic formula to check the feasibility of a schedule sensitive branch. `Ptracer` uses Z3 [10] to check the satisfiability of the generated formula. Our tool will be released as open source.

**Benchmarks.** We evaluate the performance overheads with our prototype with thirteen TBB applications, which include five TBB-based applications from Parsec [3], five geometry and graphics applications from the problem based benchmark suite (PBBS) [42], and three applications from the Structured Parallel Programming book [29]. The PBBS applications were originally implemented using Cilk [17]. We translated these applications to use Intel TBB for task parallelism. Table 1 lists the applications used and their important features.

**Evaluation environment.** The experiments were performed on a 4.00GHz four-core Intel x86-64 i7 processor, with 64 GB of memory running 64-bit Ubuntu 14.04.3. Each benchmark was executed five times and the reported performance overhead is calculated by taking the average of the five executions. We use geometric mean to report average slowdown in our evaluation.

**Effectiveness in detecting data races.** To test the effectiveness of our prototype in detecting races, we have built a test suite of 120 unit tests that include racy and non-racy programs with and without locks and schedule sensitive branches. There were 60 races in total



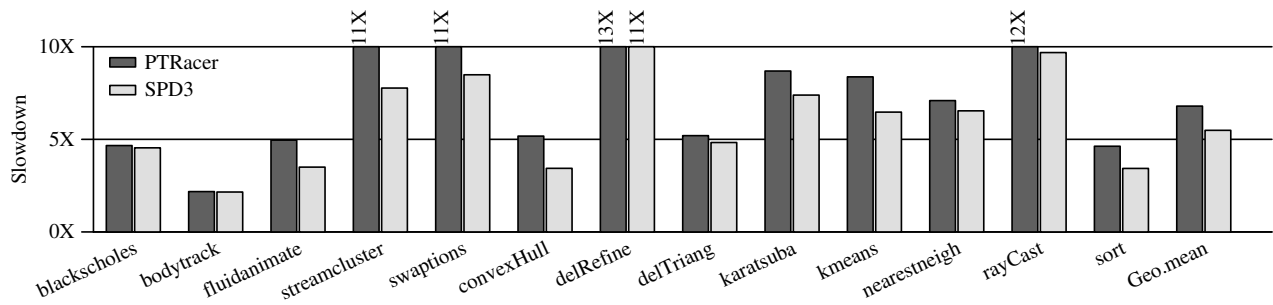


Figure 8: Execution time slowdown of PTRacer and SPD3 when compared to a baseline without any instrumentation.

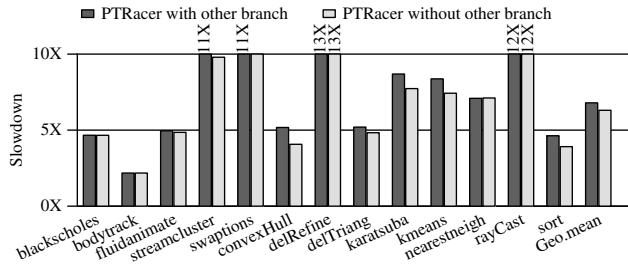


Figure 9: Execution time slowdown of PTRacer with and without data race detection with record instrumentation from the not-taken path of a branch statement.

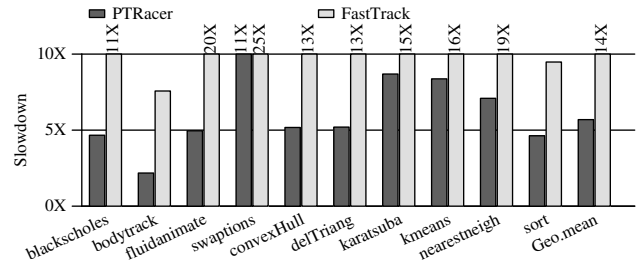


Figure 10: Execution time slowdown of PTRacer and FastTrack when compared to a baseline without any instrumentation.

in the test suite. PTRacer successfully detects all the races in the racy suite without any false positives. In contrast, SPD3 detected 46 races, reported 40 false positives, and missed 14 races. We ran SPD3 multiple times to detect races in the presence of schedule sensitive branches. There were 14 races that were not detected by SPD3 even after performing multiple executions.

**Performance overhead in comparison to SPD3.** Figure 8 reports the performance overhead of PTRacer and SPD3 when compared to a baseline without any instrumentation. We use LCA caching by default even with SPD3. There are two bars for each benchmark (smaller bars are better as it reports overheads). The average performance overhead of PTRacer is  $6.7\times$ . Although the average overhead is  $6.7\times$ , four applications, *streamcluster*, *swaptions*, *delRefine* and *raycast* have overhead greater than  $10\times$ . Among them, *streamcluster* and *swaptions* have large number of race detection checks and they perform a large number of LCA queries. *delRefine* and *raycast* have relatively fewer accesses but have no locality in their LCA queries and race checks (see Table 1). The performance overhead of SPD3 is  $5.4\times$ . SPD3 maintains constant metadata for every shared memory location and does not detect data races in programs that use locks. PTRacer has similar overheads when compared to SPD3 and detects more races in the presence of locks and schedule sensitive branches.

**Performance overhead in comparison to FastTrack.** Figure 10 reports the performance overhead of PTRacer and our implementation of FastTrack for tasks when compared to a baseline without any instrumentation. FastTrack aborted with out-of-memory errors with three applications, *streamcluster*, *delRefine* and *raycast*. These applications create many tasks and have a large number of shared accesses that prevent the use of optimized version of vector clocks. The average performance overhead of FastTrack for other

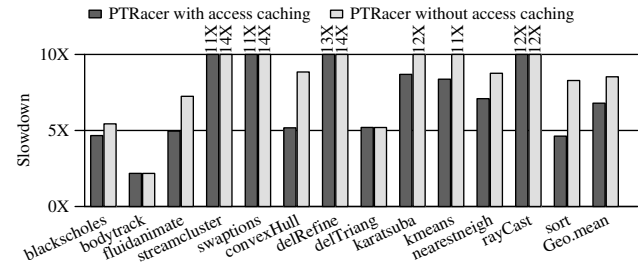


Figure 11: Execution time slowdown of PTRacer with and without access caching.

applications is  $14\times$ .

**Performance impact of record instrumentation from the not-taken path.** Figure 9 reports the performance overhead of PTRacer with and without the record instrumentation to capture operations from both the paths of a branch statement. The record instrumentation has increased the overhead of data race detection from  $6.3\times$  to  $6.7\times$ . With a nominal increase in performance overhead, the record instrumentation enables detection of races in the presence of schedule sensitive branches.

**Performance benefits with access caching.** Figure 11 reports the effect of the access caching optimization on the performance overhead of PTRacer. The access caching optimization reduces the average performance overhead from  $8.5\times$  without access caching to  $6.7\times$ . Most applications benefit from access caching. Three applications, *fluidanimate*, *convexhull*, and *sort* have significant reduction in overhead because a large fraction of the accesses hit in the access cache and do not perform the costly data race check and

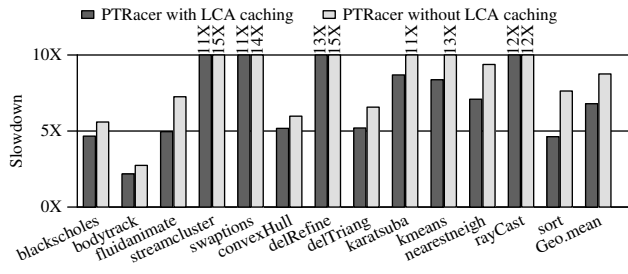


Figure 12: Execution time slowdown of `PTRacer` with and without LCA caching.

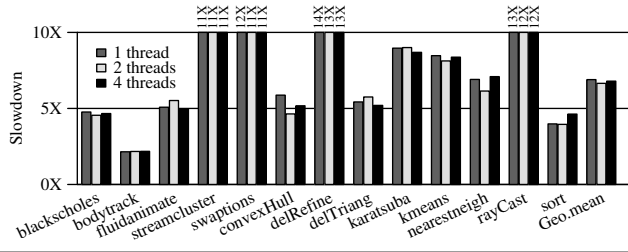


Figure 13: Execution time slowdown of `PTRacer` when executed with 1, 2 and 4 threads.

the metadata update. Applications *bodytrack*, *delrefine*, *deltriang*, and *raycast*, do not benefit from access caching since most locations are accessed once in a given step node.

**Performance benefits with LCA caching.** Figure 12 reports the performance overhead of `PTRacer` with and without LCA caching when compared to a baseline without any instrumentation. On average, LCA caching reduces the performance overhead from  $8.8\times$  without LCA caching to  $6.7\times$ . All applications except *raycast* see a significant reduction in overheads as there are fewer unique LCA queries. *raycast* does not benefit much from LCA caching because of the large number of unique LCA queries.

**Performance overhead for different number of threads.** Figure 13 reports the performance overhead of `PTRacer` when executed by restricting the task parallel runtime to use 1, 2, and 4 threads. The average overhead when executed with 1, 2, and 4 threads is  $6.8\times$ ,  $6.6\times$  and  $6.7\times$  respectively. The average overhead is almost constant which indicates that our approach scales well when the program is run on machines with larger thread/core counts.

## 6. RELATED WORK

**Data race detection in multithreaded programs.** There is a large body of work on dynamic data race detection in thread based parallel programs [12, 16, 34, 41, 48]. `FastTrack` [16] represents the state-of-the-art in dynamic data race detection for threaded programs. `FastTrack` detects data races in the given execution by tracking happens-before relations between shared memory accesses. The overhead of race detection with `FastTrack` can be high when executed on programs that create many threads as the metadata is proportional to the number of threads. Further, `FastTrack` only detects races that occur in a given schedule. `Eraser` [40] uses a lockset-based approach and checks for errors in locking discipline.

Lockset-based approaches often have lower overhead compared happens-before based approaches, but can report false positives.

There are dynamic approaches [5, 11, 14, 18, 28] that attempt to reduce the overhead of race detection through sampling. But these approaches often miss data races while trying to reduce the overhead. There are several proposals for static race detection [1, 13, 32, 35]. While static detection approaches are appealing as they have no runtime performance overhead, they can produce a large number of false positives.

**Predictive testing for threaded programs.** There are also numerous approaches that attempt to detect races and other concurrency errors feasible in a different schedule derived from a trace in a multithreaded program [20, 21, 43, 45, 47]. Predictive testing ideally can detect feasible races for a given input as long as all operations that can occur in different thread schedules occur in the observed trace. However, most predictive testing techniques bound the instruction window (up to 4K instructions) to make it practical. `PTRacer` provides guarantees similar to an ideal predictive testing by leveraging the structure of the task parallel execution and by maintaining appropriate metadata. Further, it detects races that can occur in other schedules in the presence of schedule sensitive branches using record instrumentation.

**Data race detection in task parallel programs.** The approach proposed by Mellor-Crummey *et al.* [30] and Nondeterminator [15] were seminal in proposing the detection of apparent data races in task parallel programs using the series-parallel execution graph. Subsequently, these techniques have been enhanced to handle locks [8], to handle task graphs in `Habanero-Java` [37], and to detect races without serial execution with `SPD3` [36, 38]. Our proposed research uses the `DPST` representation in `SPD3` and is inspired by the access histories in the `ALL-SETS` algorithm for `Cilk` [8].

**Determinacy checkers.** In the absence of synchronization, data race freedom ensures determinism [4, 26]. Even in deterministic programs there can be a large number of schedules for different inputs. There are proposals that memoize past schedules [9] and limit the execution to a set of input covering schedules [2]. `Tardis` [26] checks for determinism by maintaining a log of accesses and identifying conflicting accesses between tasks. In contrast, our approach detects data races both in the presence and absence of synchronization operations.

## 7. CONCLUSION

This paper addresses the problem of detecting apparent data races in task parallel programs with a parallel detector that handles locks. The key insight is to leverage the execution graph of a task parallel program to determine if accesses can occur in parallel and design metadata that tracks a constant number of access histories for each lockset held before an access to a shared memory location. `PTRacer` uses static analysis and instrumentation to identify operations that can happen in the not-taken path in the presence of schedule sensitive branches and detects apparent races that can occur in other schedules for a given input. In summary, `PTRacer` is a data race detector for task parallel programs that (a) runs in parallel, (b) detects data races when the program uses locks, (3) maintains per-location metadata that is independent of the number of dynamic accesses, and (4) detects apparent races not only in the observed schedule but also in other schedules for a given input.

## References

- [1] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for java. *ACM Trans. Program.*

- Lang. Syst.*, 28(2):207–255, 2006.
- [2] T. Bergan, L. Ceze, and D. Grossman. Input-covering schedules for multithreaded programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, pages 677–692, 2013.
- [3] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, 2008.
- [4] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, pages 97–116, 2009.
- [5] M. D. Bond, K. E. Coons, and K. S. McKinley. Pacer: Proportional detection of data races. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 255–268, 2010.
- [6] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-java: The new adventures of old x10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, pages 51–61, 2011.
- [7] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 519–538, 2005.
- [8] G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark. Detecting data races in cilk programs that use locks. In *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures*, pages 298–309, 1998.
- [9] H. Cui, J. Wu, C.-C. Tsai, and J. Yang. Stable deterministic multithreading through schedule memoization. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, pages 1–13, 2010.
- [10] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.
- [11] L. Effinger-Dean, B. Lucia, L. Ceze, D. Grossman, and H.-J. Boehm. Ifrit: Interference-free regions for dynamic data-race detection. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 467–484, 2012.
- [12] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A race and transaction-aware java runtime. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–255, 2007.
- [13] D. Engler and K. Ashcraft. Racerox: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 237–252, 2003.
- [14] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, 2010.
- [15] M. Feng and C. E. Leiserson. Efficient detection of determinacy races in cilk programs. In *Proceedings of the 9th ACM Symposium on Parallel Algorithms and Architectures*, pages 1–11, 1997.
- [16] C. Flanagan and S. N. Freund. Fastrack: Efficient and precise dynamic race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 121–133, 2009.
- [17] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 212–223, 1998.
- [18] J. L. Greathouse, Z. Ma, M. I. Frank, R. Peri, and T. Austin. Demand-driven software race detection using hardware performance counters. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, pages 165–176, 2011.
- [19] D. Grossman and R. E. Anderson. Introducing parallelism and concurrency in the data structures course. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, 2012.
- [20] J. Huang, Q. Luo, and G. Rosu. Gpredict: Generic predictive concurrency analysis. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, pages 847–857, 2015.
- [21] J. Huang, P. O. Meredith, and G. Rosu. Maximal sound predictive race detection with control flow abstraction. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 337–348, 2014.
- [22] J. Huang and L. Rauchwerger. Finding schedule-sensitive branches. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 439–449, 2015.
- [23] J. Huang, C. Zhang, and J. Dolby. Clap: Recording local executions to reproduce concurrency failures. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 141–152, 2013.
- [24] V. Kahlon, F. Ivančić, and A. Gupta. Reasoning about threads communicating via locks. In *Proceedings of the 17th International Conference on Computer Aided Verification*, pages 505–518, 2005.
- [25] D. Lea. A java fork/join framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, pages 36–43, 2000.
- [26] L. Lu, W. Ji, and M. L. Scott. Dynamic enforcement of determinism in a parallel scripting language. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 519–529, 2014.
- [27] N. Machado, B. Lucia, and L. Rodrigues. Concurrency debugging with differential schedule projections. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 586–595, 2015.

- [28] D. Marino, M. Musuvathi, and S. Narayanasamy. Literace: Effective sampling for lightweight data-race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 134–143, 2009.
- [29] M. McCool, A. Robison, and J. Reinders. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann, 2012.
- [30] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pages 24–33, 1991.
- [31] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. Cets: Compiler enforced temporal safety for c. In *Proceedings of the 2010 International Symposium on Memory Management*, 2010.
- [32] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–319, 2006.
- [33] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, pages 74–88, 1992.
- [34] E. Pozniak and A. Schuster. Efficient on-the-fly data race detection in multithreaded c++ programs. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–190, 2003.
- [35] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: Context-sensitive correlation analysis for race detection. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 320–331, 2006.
- [36] R. Raman. *Dynamic Data Race Detection for Structured Parallelism*. PhD thesis, Rice University, 2012.
- [37] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav. Efficient data race detection for async-finish parallelism. In *Proceedings of the 1st International Conference on Runtime Verification*, pages 368–383, 2010.
- [38] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav. Scalable and precise dynamic datarace detection for structured parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 531–542, 2012.
- [39] J. Reinders. *Intel Threading Building Blocks*. O’Reilly & Associates, Inc., 2007.
- [40] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 27–37, 1997.
- [41] K. Serebryany and T. Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, pages 62–71, 2009.
- [42] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. Brief announcement: The problem based benchmark suite. In *Proceedings of the 24th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 68–70, 2012.
- [43] A. Sinha, S. Malik, and A. Gupta. Efficient predictive analysis for detecting nondeterminism in multi-threaded programs. In *Formal Methods in Computer-Aided Design*, pages 6–15, 2012.
- [44] A. Sinha, S. Malik, C. Wang, and A. Gupta. Predictive analysis for detecting serializability violations through trace segmentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 446–455, 2011.
- [45] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan. Sound predictive race detection in polynomial time. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 387–400, 2012.
- [46] J. W. Vounq, R. Jhala, and S. Lerner. Relay: Static race detection on millions of lines of code. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 205–214, 2007.
- [47] C. Wang, S. Kundu, M. Ganai, and A. Gupta. Symbolic predictive analysis for concurrent programs. In *Proceedings of Formal Methods*, pages 256–272, 2009.
- [48] C. Wang, Y. Yang, A. Gupta, and G. Gopalakrishnan. Dynamic model checking with property driven pruning to detect race conditions. In *Proceedings of the 6th International Symposium on Automated Technology for Verification and Analysis*, pages 126–140, 2008.