

Atomicity Violation Checker for Task Parallel Programs

Adarsh Yoga Santosh Nagarakatte

Rutgers University, USA

{adarsh.yoga,santosh.nagarakatte}@cs.rutgers.edu



Abstract

Task based programming models (*e.g.*, Cilk, Intel TBB, X10, Java Fork-Join tasks) simplify multicore programming in contrast to programming with threads. In a task based model, the programmer specifies parallel tasks and the runtime maps these tasks to hardware threads. The runtime automatically balances the load using work-stealing and provides performance portability. However, interference between parallel tasks can result in concurrency errors.

This paper proposes a dynamic analysis technique to detect atomicity violations in task parallel programs, which could occur in different schedules for a given input without performing interleaving exploration. Our technique leverages the series-parallel dynamic execution structure of a task parallel program to identify parallel accesses. It also maintains access history metadata with each shared memory location to identify parallel accesses that can cause atomicity violations in different schedules. To streamline metadata management, the access history metadata is split into global metadata that is shared by all tasks and local metadata that is specific to each task. The global metadata tracks a fixed number of access histories for each shared memory location that capture all possible access patterns necessary for an atomicity violation. Our prototype tool for Intel Threading Building Blocks (TBB) detects atomicity violations that can potentially occur in different interleavings for a given input with performance overheads similar to Velodrome atomicity checker for thread based programs.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Software/Program Verification; D.2.5 [Software Engineering]: Testing and Debugging

Keywords Concurrency, Atomicity Checking, Intel TBB, Fork Join Programs, Debugging

1. Introduction

Programming with tasks is an abstraction for developing performance portable programs for multicore processors. In a task based programming model, the programmer specifies tasks and the runtime maps these tasks to underlying hardware threads. Task based models facilitate performance portable code (*i.e.*, scalable performance with the increase in the number of hardware cores) because the runtime balances the load with work stealing. Some popular examples of languages and frameworks for writing task parallel programs are Cilk [14], X10 [7], Java fork-join extensions [16], Intel Threading Building Blocks [29], and Habanero Java [6].

Interference between two tasks executing in parallel can cause errors such as data races and atomicity violations similar to thread-based programs. A data race exists between two parallel tasks if two tasks can logically execute in parallel, access the same location, and at least one of the accesses is a *write*. Some of these data races can violate the programmer intended atomicity specifications. The use of synchronization in tasks (*e.g.*, to avoid races) can also result in atomicity violations even when the program is free of data races. In this paper, we focus our attention on detecting atomicity violations in programs with and without data races.

A common specification of atomicity is conflict serializability [3, 12]. The execution trace of operations performed by parallel tasks is conflict serializable if the trace can be transformed into an equivalent serial trace by commuting adjacent non-conflicting operations of parallel tasks. A serial trace is obtained by concatenating the traces of the individual tasks [1]. In task-based programs, programmers expect regions of code without any task management constructs to be conflict serializable.

Our approach is motivated by prior research on data race detectors for fork-join programs [9, 10, 22, 27, 28], which detect data races for a given input without performing interleaving exploration. They identify simultaneous read and write operations executed by distinct tasks that can execute in parallel using the dynamic series-parallel execution graph. These techniques detect all data races for a given input in the absence of locks or critical sections. When tasks use locks, these techniques detect races in a given trace. Nondeterminator-2 [9] detects races for a given input in the presence of locks for a class of programs with commutative

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in the following publication:

CGO '16, March 12–18, 2016, Barcelona, Spain
© 2016 ACM. 978-1-4503-3778-6/16/03...
<http://dx.doi.org/10.1145/2854038.2854063>

critical sections (Abelian programs). It maintains a history of all dynamic accesses performed by parallel tasks and checks this history on each access to determine if a race is feasible in the current trace and in other schedules [9].

Inspired by these approaches for race detection, our goal is to detect atomicity violations by designing appropriate metadata and by leveraging the dynamic execution structure of a task parallel execution. Given a program with atomicity annotations, we propose to maintain a history of dynamic memory accesses performed by various tasks with each shared memory location. Our proposed technique determines if memory accesses performed by potentially parallel tasks are conflict serializable using the access history. The access history can be used to determine if there exists two accesses by the same task and an interleaving access by a parallel task that results in an atomicity violation. We also observe that it is not necessary to maintain access history for every dynamic memory access. We show that atomicity violations, both in an execution trace and in other possible schedules for a given input, can be detected by maintaining a small number of access history entries that capture all possible ways in which an atomicity violation can occur.

To streamline the implementation of the metadata space, we split the access history metadata with each shared memory location between a global metadata space and a local metadata space. The global metadata space maintains twelve access history entries for each shared memory location. Among these twelve access history entries, eight of them capture the four different kinds of two-access patterns performed by tasks to a shared memory location (*i.e.*, read-read, read-write, write-write, and write-read operations performed by the same task). The atomicity of these two-access patterns can be violated by an interleaving parallel access by a different task. To track such accesses, the other four access history entries capture the two distinct read operations and two distinct write operations performed by different tasks that can execute in parallel. The local metadata space with each task maintains the first read and first write to a location by that task. When a task performs more than one memory access to a memory location, the access history entries corresponding to these accesses — one from the local metadata space entry and other for the current access — are updated in the global metadata space if that access pattern has not already been recorded in the global space (see Section 3.2). Hence, the local metadata space acts as an interim buffer to hold information about the first access by a task until it performs the second access.

To detect atomicity violations with locks, the access history entry in the local metadata space also maintains information about the set of locks held by a task before the memory access. We also provide a unique name to a lock when it is released and re-acquired by the same task (see Section 3.3). The access history corresponding to two accesses performed by a task is updated in the global space only when

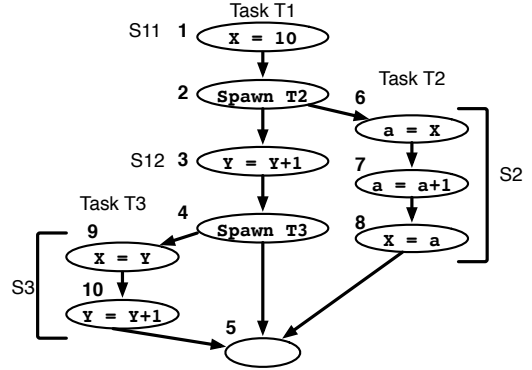


Figure 1: A task parallel program with three tasks, T1, T2, and T3. The regions of code without any synchronization constructs are labeled S11, S12, S2, and S3.

the intersection of the set of locks held by the two accesses is empty (*i.e.*, the accesses occur in two different critical sections).

The proposed approach detects both single- and multiple-variable atomicity violations. When multiple locations are required to be accessed atomically, our approach provides the same metadata to all those locations. The proposed technique is sound (*i.e.*, it detects all atomicity violations in a given trace) and is precise (*i.e.*, it reports no false positives). Our technique is complete, provided the execution trace observed by the dynamic analysis contains all shared memory operations that can possibly occur in other interleavings for a given input.

We have built a prototype tool for detecting atomicity violations for task parallel programs written with the Intel Threading Building Blocks (TBB) library. Our technique successfully detects all errors in our test suite exercising various kinds of atomicity violations. The prototype’s performance overheads are comparable to the Velodrome atomicity checker for threaded programs [12], which detects atomicity violations in a given trace. In contrast to Velodrome, our technique detects atomicity violations that could possibly occur in different interleavings for a given input.

2. Dynamic Program Structure Tree

In this section, we describe prior research on using the execution structure of a task parallel program to check if two accesses can logically occur in parallel [28], which we use in our approach. We say two accesses by different tasks can logically execute in parallel if there exists a schedule where these accesses are performed in parallel even though the observed execution does not have a parallel access.

Consider the example task parallel program in Figure 1. We illustrate task creation with the `spawn` statement. Task T1 creates two tasks T2 and T3. Prior research on data race detection for fork-join programs determined that the exe-

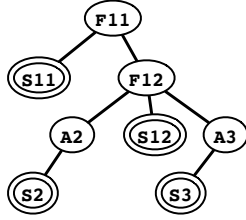


Figure 2: DPST for the sample task parallel program in Figure 1 after it has executed all the statements. There are four step nodes (S11, S12, S2, S3) in total in the DPST. The step nodes S11 and S12 belong to task T1. The step nodes S2 and S3 belong to tasks T2 and T3, respectively. The step node S2 is a child of async node A2 because S2 is executed by task T2. Similarly, the step node S3 is a child of async node A3 because S3 is executed by task T3. The step nodes S2 and S3 can occur in parallel because the LCA(S2, S3) is F12 and the left child of F12 is an **async** node and it is an ancestor of S2.

cution of a task parallel program results in a series parallel graph [9, 10, 22], which can be used to find races in other schedules not encountered in the current execution trace. Unfortunately, the program has to be executed serially, which results in performance overheads.

Raman *et al.* [28] proposed an approach to check whether two accesses can logically execute in parallel without resorting to a serial execution. Their key contribution is in representing the execution as an ordered tree, which is called the Dynamic Program Structure Tree (DPST), to capture the series-parallel relationships among tasks. DPST can handle both spawn-sync constructs in Cilk/Intel TBB and async-finish constructs in Habanero Java [26, 28]. Two accesses can execute in parallel if the least common ancestor of the nodes corresponding to the two accesses have certain properties, which we describe below.

The DPST has three kinds of nodes. First, a maximal sequence of instructions executed by a task without any task management constructs is represented by a **step** node. All computation and data accesses occur in the step nodes. Hence, every data access has a corresponding step node associated with it. These step nodes are always leaves in a DPST. Second, **async** nodes capture the spawning of a task by a parent task. An async node executes asynchronously with the remainder of the parent task. Third, 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. All siblings of a node in a DPST are ordered from left to right to reflect the left-to-right sequencing of computations of their parent task.

The DPST is constructed such that all inner nodes are either **async** or **finish** nodes and all leaf nodes are **step** nodes. Although the nodes are added to the DPST during execution,

```

procedure BASICATOMICITYCHECKER( $l, S_i, A$ )
   $AH \leftarrow Metadata(l)$ 
  for all  $p \in AH$  do
    if  $Node(p) == S_i$  then
      for all  $q \in AH \setminus p$  do
         $S_j = Node(q)$ 
         $r = \langle S_i, A \rangle$ 
        if  $Par(S_i, S_j) \&\& !Serialize(r, p, q)$  then
          Report Atomicity Violation
   $Metadata(l).append(\langle S_i, A \rangle)$ 
  return

```

Figure 3: Basic algorithm to check atomicity violations on memory access to location l by the step node S_i with access type A . $Metadata(l)$ function returns the access history associated with location l . $Node(p)$ returns the step node of access history p . $Par(S_i, S_j)$ returns true if the step nodes S_i and S_j in the DPST can execute in parallel. $Serialize(r, p, q)$ returns false when r, p , and q form an unserializable triple.

the path from a node to the root and left-to-right ordering of the siblings do not change. The DPST’s construction ensures that two distinct step nodes S_1 and S_2 (let’s assume S_1 is to the left of S_2) are in parallel if the least common ancestor (LCA) of S_1 and S_2 has an immediate child A that is an **async** node and is also an ancestor of S_1 .

Figure 2 presents the DPST after all tasks and instructions in the program in Figure 1 have executed. The step nodes S2 and S12 can occur in parallel since the LCA(S2, S12) is F12 and its left child is an **async** node. Similarly, S2 and S3 can occur in parallel. However, step nodes S11 and S2 cannot occur in parallel since the LCA(S11, S2) is F11 and its left child that is also an ancestor of S11 is S11, which is not an **async** node. Similarly, step nodes S12 and S3 cannot occur in parallel.

Next, we describe the detection of atomicity violations with appropriate metadata leveraging the DPST.

3. Our Approach

We propose an approach to detect atomicity violations that can occur in other schedules for a given input by maintaining appropriate metadata and by leveraging the DPST. In a task parallel program, some atomicity violations are necessary (*e.g.*, spin based synchronization operations). Hence, the programmer provides annotations to specify the memory location (or a group of memory locations) that needs to be accessed atomically by a task. Initially, we describe the basic approach (Section 3.1) for detecting atomicity violations in task parallel programs that do not use synchronization operations. In such scenarios, locations involved in atomicity violations are also data races. However, all data races are not atomicity violations. We optimize the basic approach by splitting the access history metadata into global metadata and local metadata (Section 3.2) and by handling synchronization operations (Section 3.3).

Access Pattern	Conflict Serializable?	Access Pattern	Conflict Serializable?
A1:Read A2:Read A3:Read	Serializable	A1:Read A2:Read A3:Write	Serializable
A1:Read A2:Write A3:Read	Un-serializable	A1:Read A2:Write A3:Write	Un-serializable
A1:Write A2:Read A3:Read	Serializable	A1:Write A2:Read A3:Write	Un-serializable
A1:Write A2:Write A3:Read	Un-serializable	A1:Write A2:Write A3:Write	Un-serializable

Figure 4: Access patterns that are not conflict serializable. Accesses A1 and A3 are performed by the same step node in a task and A2 is performed by a step node in a parallel task.

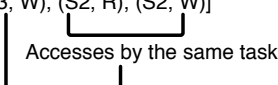
3.1 Basic Approach

To detect potential atomicity violations that can occur in different schedules for a given input without interleaving exploration, our approach needs to (1) identify tasks that can logically execute in parallel, (2) identify multiple accesses to the same location by a task, and (3) identify an interleaving access from a task that can logically execute in parallel (even though the current trace may not exhibit such an access) and results in a trace that is not conflict serializable. Figure 3 provides the basic algorithm for detecting atomicity violations. We construct the dynamic program structure tree (DPST) when the task parallel program executes as described in Section 2. We perform a least common ancestor query on the DPST to determine if two accesses can logically execute in parallel.

To detect atomicity violations that can possibly occur in different schedules, our approach maintains access histories for each memory location. The access history associated with each location is a list of entries where each entry contains the step node performing the access and the access type. On a memory access to a location l by step node S_i , we check if there is a prior access performed by the same step node along with another access by a logically parallel step node S_j , which results in an atomicity violation.

Conflict serializability. Building on prior work [12], we detect atomicity violations by checking whether the access history violates conflict serializability. Given three accesses (A1, A2, and A3) in the access history, where A1 and A3 are performed by the same step node and A2 is performed by a step node executing in parallel, we check whether these three accesses are conflict serializable. The three accesses are conflict serializable if we can obtain a serial trace by reordering non-conflicting operations [1]. Two accesses are in conflict if and only if they access the same memory location, they belong to two different step nodes in distinct tasks, and at least one of them is a write. Figure 4 identifies the access patterns involving three accesses that are conflict serializable [19].

Execution Trace	Metadata for X
1	[(S11, W)]
2	[(S11, W)]
3	[(S11, W)]
4	[(S11, W)]
9	[(S11, W), (S3, W)]
10	[(S11, W), (S3, W)]
6	[(S11, W), (S3, W), (S2, R)]
7	[(S11, W), (S3, W), (S2, R)]
8	[(S11, W), (S3, W), (S2, R), (S2, W)]



 Three accesses that form an unserializable triple

Figure 5: Execution trace of the program in Figure 1. The access history metadata for memory location X is shown after executing each statement in the trace. Although the trace does not have an atomicity violation, the access history metadata enables the detection of atomicity errors that can occur in a different schedule. The step nodes S2 and S3 can execute in parallel.

Figure 3 presents the algorithm to check atomicity violations on access to location l . It identifies two other prior accesses (p and q) that access the same location, where one of them is executed by the same step node and the other is executed by another parallel step node. It detects atomicity violations by checking conflict serializability of the current access with two prior accesses.

Figure 5 illustrates the basic approach for the task parallel program in Figure 1. The programmer specifies that all accesses to memory location X in a step node should be atomic. The trace observed during the execution is shown in Figure 5, which does not have atomicity violations as all operations of the step nodes S2 and S3 are executed together. However, our access history metadata keeps track of all dynamic accesses and detects that there are two operations in the step node S2 that access the same memory location, S3 also accesses the same location, and these three accesses form an unserializable triple.

Restrictions. Our approach detects all atomicity violations for a given input (*i.e.*, complete) provided the execution trace has all the shared memory operations (possibly in different orders) executed by different interleavings. This requirement can be violated when (1) a conditional branch or the structure of the execution graph depends on a racy access and (2) conditional branches are present within blocks of code protected by synchronization.

3.2 Metadata Organization for Optimized Detection

The access history metadata in the basic approach described earlier is proportional to the number of dynamic memory accesses, which is checked on every memory access. Further, different locations will have different amounts of meta-

```

procedure OPTATOMICITYCHECKER( $l, S_i, A$ )
   $t \leftarrow \text{Task}(S_i)$ 
  if  $GS(l) == \emptyset$  and  $LS_t(l) == \emptyset$  then
     $\text{HandleFirstAccess}(l, S_i, A)$ 
    return
  if  $GS(l) \neq \emptyset$  and  $LS_t(l) == \emptyset$  then
     $\text{HandleFirstAccessCurrentTask}(l, S_i, A)$ 
    return
  if  $GS(l) \neq \emptyset$  and  $LS_t(l) \neq \emptyset$  then
     $\text{HandleNonFirstAccess}(l, S_i, A)$ 
    return

```

Figure 6: High level structure of atomicity checking with fixed size metadata. On each access to location l by task t , we check whether it is the first access to the global metadata space (GS) or the local metadata space LS_t and take appropriate actions. The algorithms for `HandleFirstAccess`, `HandleFirstAccessCurrentTask`, and `HandleNonFirstAccess` are presented in Figure 7, Figure 8, and Figure 9, respectively.

data, which necessitates dynamic allocation and complicated management of the metadata space. We make the observation that it is not necessary to maintain access history information for all dynamic memory accesses to a location. We can detect atomicity violations when we capture enough information to reason about all unserializable triples shown in Figure 4. An atomicity violation involves three accesses: two accesses from a single task and one interleaving access from a parallel task. Hence, we propose to maintain access history metadata that remembers distinct two-access patterns (read-read, read-write, write-read, and write-write operations by a task) with every shared memory location. The atomicity of the two-access pattern can be violated by an interleaving parallel access. Hence, we propose to maintain single-access patterns (*i.e.*, read and write operations) along with the two-access patterns as metadata.

3.2.1 Global and Local Metadata

To streamline the implementation of the access history metadata, we split it into global metadata and local metadata. The global metadata for each memory location is shared by all tasks. The global access history metadata contains twelve access history entries for each shared memory location, corresponding to the four distinct two-access patterns (read-read, read-write, write-read, and write-write operations performed by the same task) along with access histories for the two distinct read and two distinct write operations. Each access history entry contains information about the step node and the type of the access. We use the following notation to refer to single-access patterns in the global metadata space for location l : $GS(l).R_1$, $GS(l).R_2$, $GS(l).W_1$, and $GS(l).W_2$. We refer to the two-access patterns with the following notation: $GS(l).RR$ (read-read), $GS(l).RW$ (read-write), $GS(l).WW$ (write-write), and $GS(l).WR$ (write-read).

```

procedure HANDLEFIRSTACCESS( $l, S_i, A$ )
   $t \leftarrow \text{Task}(S_i)$ 
  if  $A == \text{read}$  then
     $GS(l).R_1 = \langle S_i, \text{read} \rangle$ 
     $LS_t(l).R = \langle S_i, \text{read} \rangle$ 
  else
     $GS(l).W_1 = \langle S_i, \text{write} \rangle$ 
     $LS_t(l).W = \langle S_i, \text{write} \rangle$ 
  return

```

Figure 7: This algorithm updates the global metadata (GS) and local metadata (LS) on first access to location l by any task with access type A , which is either a read or a write.

In addition to the global access history metadata, each task also maintains per-task local metadata that maintains information about its first read and write operation to a location. When multiple parallel tasks have performed one access to a location, we do not know if these tasks will perform another access to the same location in the future that would form a two-access pattern. The local metadata space acts as an interim buffer to hold information about the first access until we find the second access. We update the global metadata space when we find a two-access pattern if that access-pattern has not already been encountered or the access history entry for the pattern in the global space is performed by a task that executes in series with the current task. In summary, we maintain global metadata (GS) that stores twelve access histories and per-task local metadata (LS_t) that stores two access histories with each location.

3.2.2 Metadata Propagation and Checking

In this section, we describe when the global and the local metadata space is updated and checked. Figure 6 provides a high level overview of our algorithm. It updates and checks metadata depending on whether the access is: (1) the first access to a location by any task, (2) the first access by the current task but has been accessed by other tasks, or (3) is a repeated access to a memory location that has been already accessed by the current task. We will use Figure 10 as a running example to illustrate metadata updates and checks.

First access. When a shared memory location is accessed for the first time by any task and there is no entry either in the local or the global space, the algorithm updates the single-access pattern in the global metadata space and the first read/write information in the local metadata space as shown in Figure 7. In Figure 10, statement 1 in step node S11 performs the first access to location X . Hence, the single-access pattern in the global metadata space (*i.e.*, $GS(X).W_1$) and the first write entry in the local metadata space of task T1 is updated as shown in Figure 10.

First local access with prior accesses by other tasks. In the second case, the location l has been accessed by other tasks but is the first access by the current task. Hence, the

```

1: procedure HANDLEFIRSTACCESSCURRENTTASK( $l, S_i, A$ )
2:    $t \leftarrow Task(S_i)$ 
3:   if  $A == \text{read}$  then
4:      $LS_t(l).R = \langle S_i, \text{read} \rangle$ 
5:      $Check(GS(l).WW, LS_t(l).R)$ 
6:     if  $GS(l).R_1 == \emptyset$  or  $!Par(GS(l).R_1, S_i)$  then
7:        $GS(l).R_1 = \langle S_i, \text{read} \rangle$ 
8:     else if  $GS(l).R_2 == \emptyset$  or  $!Par(GS(l).R_2, S_i)$  then
9:        $GS(l).R_2 = \langle S_i, \text{read} \rangle$ 
10:  else
11:     $LS_t(l).W = \langle S_i, \text{write} \rangle$ 
12:     $Check(GS(l).WW, LS_t(l).W)$ 
13:     $Check(GS(l).RW, LS_t(l).W)$ 
14:     $Check(GS(l).RR, LS_t(l).W)$ 
15:     $Check(GS(l).WR, LS_t(l).W)$ 
16:    if  $GS(l).W_1 == \emptyset$  or  $!Par(GS(l).W_1, S_i)$  then
17:       $GS(l).W_1 = \langle S_i, \text{write} \rangle$ 
18:    else if  $GS(l).W_2 == \emptyset$  or  $!Par(GS(l).W_2, S_i)$  then
19:       $GS(l).W_2 = \langle S_i, \text{write} \rangle$ 
20:  return

```

Figure 8: Metadata updates and checks on the first access by step node S_i to location l that has been accessed by other tasks. We check if the current access can cause atomicity violations with two access patterns in the global space. We update the global and the local metadata space as shown above. The `Check` function takes access histories corresponding to a two-access pattern and the current access and checks if they form an unserializable triple. The `Par` function checks if the two step nodes corresponding to the access histories of its arguments can occur in parallel.

local metadata space does not have an entry for location l but there is an access history entry in the global metadata space. The only way the current access can result in an atomicity violation is if it forms an unserializable triple with the two-access patterns in the global metadata space.

If the current access is a read operation, then we check whether the access histories corresponding to the write-write access pattern in the global metadata space form an unserializable triple with the current access (see line 5 in Figure 8). We also check if the current access occurs in parallel with the single-access reads in the global metadata space. If the current access is in series with the single-access pattern (R_1 or R_2) in the global metadata space, then it is updated with the current access (see lines 6-9 in Figure 8).

If the current access is a write operation, then we check whether the access histories corresponding to the read-write, write-write, write-read, and read-read two-access patterns in the global space form an unserializable triple with the current access (see lines 12-15 in Figure 8). We also update the single-access patterns for writes (W_1 and W_2) with the current access if it is in series with the current access (see lines 16-19 in Figure 8). Finally, we update the local metadata space to reflect the current access.

In Figure 10, statement 9 in step node S3 performs the first access by task T3 to location X, which has already been

```

1: procedure HANDLENONFIRSTACCESS( $l, S_i, A$ )
2:    $t \leftarrow Task(S_i)$ 
3:   if  $A == \text{read}$  then
4:     if  $LS_t(l).R \neq \emptyset$  and  $!Par(GS(l).RR, S_i)$  then
5:        $GS(l).RR = [LS_t(l).R, \langle S_i, \text{read} \rangle]$ 
6:        $Check(GS(l).RR, GS(l).W_1)$ 
7:        $Check(GS(l).RR, GS(l).W_2)$ 
8:     if  $LS_t(l).W \neq \emptyset$  and  $!Par(GS(l).WR, S_i)$  then
9:        $GS(l).WR = [LS_t(l).W, \langle S_i, \text{read} \rangle]$ 
10:       $Check(GS(l).WR, GS(l).W_1)$ 
11:       $Check(GS(l).WR, GS(l).W_2)$ 
12:     if  $!Par(GS(l).R_1, S_i)$  or  $!Par(GS(l).R_2, S_i)$  then
13:        $GS(l).(R_1 \text{ or } R_2) = \langle S_i, \text{read} \rangle$ 
14:   else
15:     if  $LS_t(l).R \neq \emptyset$  and  $!Par(GS(l).RW, S_i)$  then
16:        $GS(l).RW = [LS_t(l).R, \langle S_i, \text{write} \rangle]$ 
17:        $Check(GS(l).RW, GS(l).W_1)$ 
18:        $Check(GS(l).RW, GS(l).W_2)$ 
19:     if  $LS_t(l).W \neq \emptyset$  and  $!Par(GS(l).WW, S_i)$  then
20:        $GS(l).WW = [LS_t(l).R, \langle S_i, \text{write} \rangle]$ 
21:        $Check(GS(l).WW, GS(l).W_1)$ 
22:        $Check(GS(l).WW, GS(l).W_2)$ 
23:        $Check(GS(l).WW, GS(l).R_1)$ 
24:        $Check(GS(l).WW, GS(l).R_2)$ 
25:     if  $!Par(GS(l).W_1 \text{ or } W_2, S_i)$  then
26:        $GS(l).(W_1 \text{ or } W_2) = \langle S_i, \text{write} \rangle$ 
27:   return

```

Figure 9: Metadata propagation and checking on an access to location l that has been previously accessed both by the current step node S_i and by other tasks. Since the current step node has performed at least two accesses, the algorithm checks if their atomicity can be violated by global access histories: R_1 , R_2 , W_1 , and W_2 using the `Check` function. We use the notation $GS(l).WW$ to refer to access histories capturing the write-write pattern in the global space for location l . The two access pattern is a list of two access history entries. We use the notation $[LS_t(l).R, \langle S_i, \text{read} \rangle]$ to refer to the new two access pattern created using the access history in the local space $LS_t(l).R$ and the access history corresponding to the current access $\langle S_i, \text{read} \rangle$.

accessed by task T1. The local metadata space of task T3 is updated with the write by step node S3. The single-access pattern in the global metadata space for location X (*i.e.*, $GS(X).W_2$) is updated with the current write.

Non-first access. In this case, the task performing the current access has already performed another access to the same location in the past. We use the information from the local metadata space to determine if there is a need to update the global metadata space. If the current operation is a read and the current task has already performed a read, the algorithm checks if the atomicity of the read-read two access pattern seen by the current step node can be violated by the single-access patterns in the global space (see lines 6-7 in Figure 9). It also updates the global read-read pattern if the entry in the global space occurs in series with the current ac-

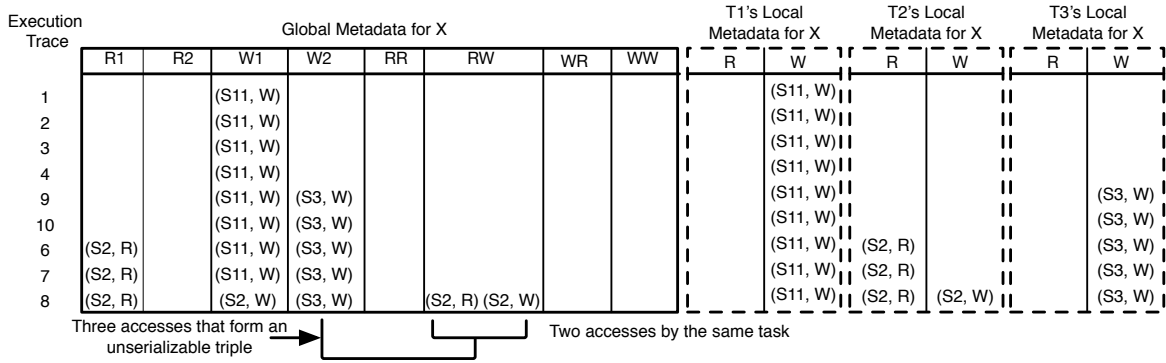


Figure 10: Atomicity checking with global and local metadata spaces for the task parallel program in Figure 1. We use the execution trace from the illustration of the basic approach in Figure 5. The global metadata space has 12 access history entries. The local metadata space has two access history entries corresponding to first read and write operations by the step node. There are 3 local metadata spaces corresponding to the three active tasks. The metadata in the global and local spaces after executing each statement in the trace is shown.

cess (see line 5 in Figure 9). Similarly, if the current task has already performed a write, the algorithm checks if the atomicity violation can occur between the write-read pattern seen by the local task and the single-access write patterns in the global space (see lines 10-12 in Figure 9). It also updates the global metadata with the write-read pattern seen by the local task if the global write-read pattern occurs in series with the current access (see line 9 in Figure 9).

If the current access by the task is a write and it has already seen a read, then the algorithm checks if the atomicity of the read-write pattern seen by the local task can be violated by single-access write pattern in the global space (see lines 17-18 in Figure 9). If the local task has already performed a write operation in the past, the algorithm checks if the atomicity of the write-write pattern in the current step node can be violated by global single-access patterns (see lines 21-24 in Figure 9). It also updates the read-write/write-write patterns in the global space if the entries in the global space are in series with the step node of the current access.

In Figure 10, access to location X by statement 8 in step node S2 is a non-first access because the step node S2 has already performed a prior read to X. The atomicity of the current access and the prior access to X by S2 is checked with the single-access write patterns in the global space, which enables the detection of the atomicity violation.

3.3 Handling Locks

Task parallel programs can also use synchronization operations (e.g., Intel TBB provides optimized implementations of various locks). Beyond data races, atomicity violations can occur in a task parallel program with locks if there are two accesses to the same shared memory location that occur in different critical sections and there is an interleaving access from another task. Figure 11 presents a data-race free version of the task parallel program from Figure 1, where accesses

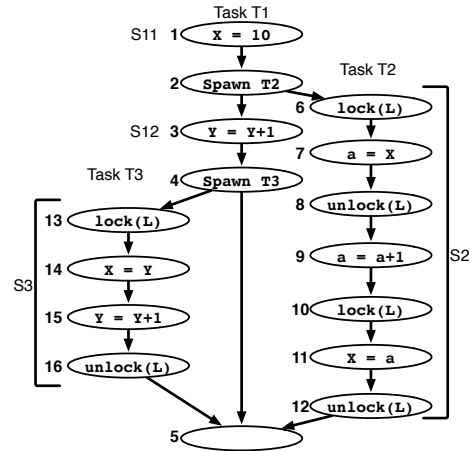


Figure 11: A revised data-race free task parallel program from Figure 1 with shared memory accesses to X in step nodes S2 and S3 protected by lock L.

to X have been protected by lock L. To detect atomicity violations in the presence of locks, we extend the algorithm described in Figure 6 in two dimensions: (1) the access history in the local metadata space keeps track of locks held by the step node at the time of the access and (2) the two-access patterns in the global metadata space are updated with local metadata entries when the two accesses performed by the step node are not protected by the same lock and they satisfy the conditions described in Figure 9. Note that the global metadata space is unchanged, as the information about locks is maintained only in the local metadata space.

Lock versioning. If a step node releases and re-acquires the same lock between two accesses to the same shared memory location, a parallel access by another task can cause

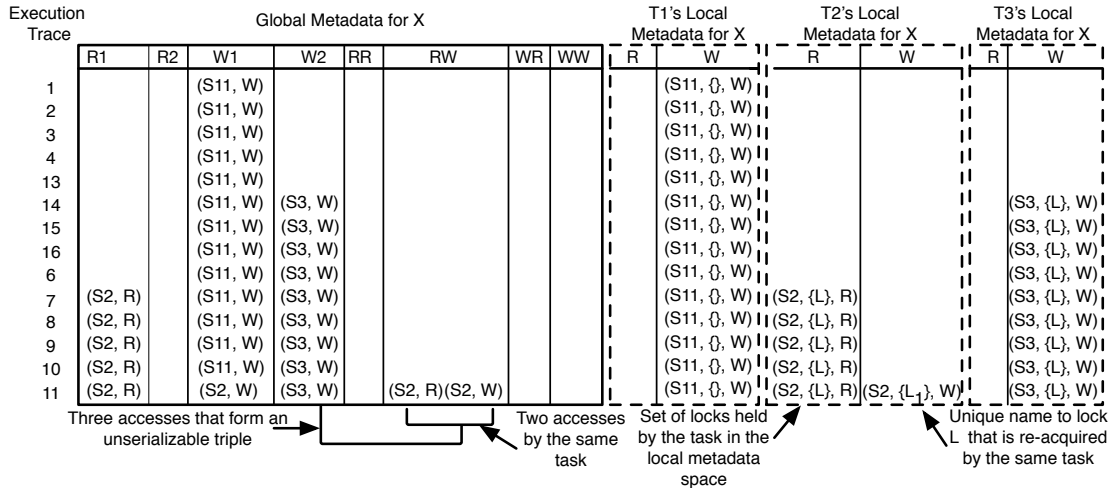


Figure 12: Detection of atomicity errors in the presence of locks. We illustrate how the local and global metadata get updated as the statements from the task parallel program in Figure 11 are executed. The local metadata space tracks the set of locks held by the task for each access. The set of locks held by task T2 when the read to X by statement 7 is executed is $\{L\}$. When the same lock L is released and reacquired by T2, we provide a unique name to the lock, L_1 . Hence, the set of locks held by T2 on the execution of statement 11 is $\{L_1\}$. The global metadata space does not contain information about the set of locks held by the access.

atomicity violations. To detect atomicity violations in such scenarios, we provide a unique name for the lock every time it is re-acquired within the same step node. If two accesses to a location are in the same critical section, then they would be protected by the same lock and would not experience an atomicity violation. Two accesses are protected by the same lock if the intersection of the set of locks held by them is not empty. When two accesses are in different critical sections, the set of locks held would differ even when the critical sections acquire and release the same lock because the re-acquired lock has a unique name.

In Figure 12, the set of locks held by each task before an access is tracked in the local metadata space. The two-access pattern is updated in the global space when the intersection of the set of locks held before the two accesses is empty, as shown in Figure 12 after the execution of statement 11.

4. Experiments

This section describes our prototype, implementation optimizations, benchmarks, and experimental evaluation to measure the performance overhead.

Prototype. Our prototype is designed for C++ programs that use Intel Threading Building Blocks (TBB) for task parallelism. It consists of an annotation processor that processes programmer-provided annotations, an instrumenter that adds calls to the checker, and the instrumentation library that performs the metadata propagation and checking. We use LLVM-3.7 with Clang as the foundation for processing annotations and adding instrumentation code. The programmer uses type qualifiers to provide annotations [13]. The front-

end processes these qualifiers and generates LLVM IR with the qualifier information encoded as LLVM metadata. We use a compiler pass on the LLVM IR to insert calls to our instrumentation library on memory accesses of interest, synchronization accesses, and task management constructs. The instrumentation functions are packaged as a library, which is linked with the executable. We also changed the Intel TBB library to add calls to our instrumentation functions on task creation, task completion, synchronization, and to pass task and thread identifiers. Our prototype is open source.¹

Implementation optimizations. To reduce the performance overhead, we optimize the layout of the DPST and cache the least common ancestor queries. Instead of using a linked data structure for the DPST, which has low data locality, our prototype overlays the DPST in a linear array of nodes. Each node in the DPST maintains an index to the parent, which avoids unnecessary pointer indirection, provides better locality, and avoids the cost of frequent dynamic allocations. We cache the frequently accessed LCA queries to reduce the overhead of repeated traversals in the DPST.

Benchmarks. We evaluate our prototype with thirteen TBB applications, which include five TBB-based applications from Parsec [2], five geometry and graphics applications from the problem based benchmark suite (PBBS) [31], and three applications from the Structured Parallel Programming book [21]. The PBBS applications were originally implemented using Cilk [14]. We translated these applications to use Intel TBB for task parallelism. Table 1 lists the bench-

¹ The latest version of our prototype can be found at <https://github.com/rutgers-apl/Atomicity-Violation-Detector>.

Benchmark	No. of locations	No. of nodes	No. of LCAs	% of unique LCAs
blackscholes	10M	1,352	0	-NA-
bodytrack	5,101	915,537	11,567	56.32
streamcluster	4.58M	530,952	234,781	49.87
swaptions	26.76M	144M	9.87M	64.41
fluidanimate	19.73M	759,830	7.41M	61.35
convexhull	6.28M	91.17M	4.31M	62.11
delrefine	9.12M	4.87M	8.19M	65.76
deltriang	20M	4.14M	97,437	61.38
karatsuba	638,282	198,379	39,836	54.55
kmeans	40M	220,788	18.29M	83.86
nearestneigh	1.13M	18.69M	539,031	53.13
raycast	3.89M	6.28M	61.48M	91.13
sort	26,984	2,443	8,165	56.67

Table 1: The table lists the number of unique dynamic memory locations accessed, the number of nodes in the DPST, the number of least common ancestor queries, and the percentage of unique LCA queries for each benchmark. We use M for million in the table. The blackscholes application uses `parallel_for` and does not perform multiple accesses to the same location in any step node. Our approach does not perform an LCA query on the first access to a location by a step node (see Figure 7) and hence, blackscholes has zero LCA queries.

marks used and the characteristics of the benchmarks.

The experiments were performed on a 2.1GHz 16-core Intel x86-64 Xeon server with 64 GB of memory running 64-bit Ubuntu 14.04.3. Each benchmark was executed five times and the overhead reported is calculated by taking the average of the five executions. We use geometric mean to report the average slowdown in our evaluation.

Detection of atomicity violations. We have built a test suite of 36 programs that exercise various kinds of atomicity violations. Our prototype detected all these violations without false positives. We have also developed a trace generator that takes the number of tasks and memory accesses as parameter and generates execution traces. Our prototype successfully detects all atomicity violations for a given input by examining one execution trace.

Experimental evaluation. We evaluate the proposed technique to measure the performance overhead and study the impact of optimizations. We also compare our prototype with Velodrome [12], which is an atomicity checker for threaded programs and detects atomicity violations that occur in a given schedule. We reimplemented it to check the atomicity of accesses performed by a step node.

Performance overhead in comparison to Velodrome. Figure 13 reports the runtime performance overhead of atomicity checking compared to a baseline without any checking. There are two bars for each benchmark. The left bar and the right bar report the performance overhead for

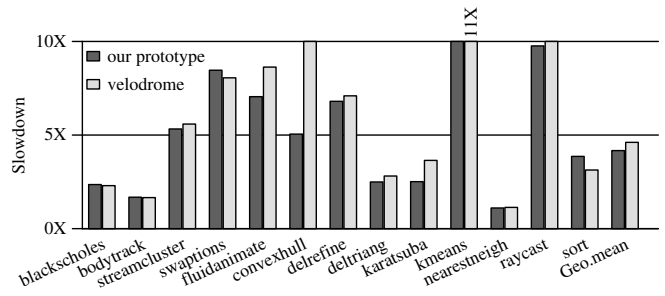


Figure 13: Execution time slowdown of our prototype atomicity violation detector and Velodrome when compared to a baseline without any instrumentation. As each bar represents slowdown, smaller bars are better.

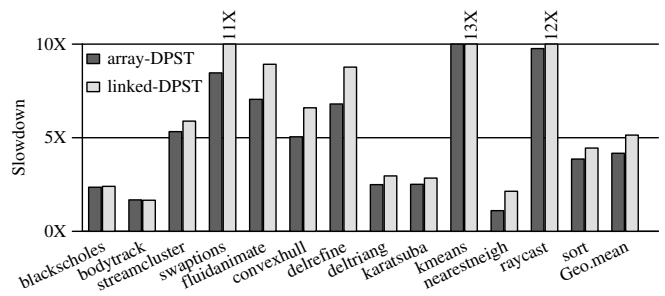


Figure 14: Execution time slowdown of atomicity checking with our prototype using an array-based DPST and a linked data structure based DPST.

our prototype and Velodrome respectively. The performance overhead for atomicity checking with our technique is $4.2\times$. Three applications, *kmeans*, *raycast*, and *swaptions* have relatively high overhead compared to other benchmarks. Among them, *kmeans* and *raycast* perform a large number of LCA queries and a large fraction of them are unique and do not benefit from LCA caching (see Table 1). Although *swaptions* performs relatively few LCA queries, it has higher overhead than other applications because it has the highest number of nodes in the DPST and accesses a large number of memory locations.

The performance overhead of atomicity checking for a given schedule with Velodrome is $4.6\times$ on average. As Velodrome detects atomicity violation in a given schedule, it has to be combined with an interleaving explorer to detect atomicity violations possible in other schedules. Our approach detects atomicity violation with similar or lower overhead than Velodrome while providing detecting atomicity violations that can occur in other schedules for a given input.

Array-based DPST vs linked DPST. Figure 14 reports the performance overhead of our prototype with DPST implemented as a linear array of nodes and DPST imple-

mented with linked data structures. On average, the array-based DPST reduces the performance overhead of atomicity checking from $5.1\times$ with linked DPST to $4.2\times$. Applications that have a large number of LCA queries benefit from array-based DPST.

5. Related Work

Atomicity checkers for threaded programs. There is huge body of work on detecting atomicity violations in threaded programs [3, 12, 18–20, 25], which detect violations in a trace. Many of these techniques use conflict serializability [3, 12]. They expose atomicity violations by perturbing executions [25], actively looking for races [11], inferring atomicity specifications [18] and identifying correlation between multiple variables [18]. To detect atomicity violations in other schedules for a given input, these techniques should be used in tandem with interleaving exploration strategies [5, 23, 24].

Static analysis and predictive testing. Several static analysis techniques detect atomicity violations through variants of type inference [30]. The false positives from static analysis have also been filtered by complementing static analysis with dynamic analysis [8]. Predictive approaches attempt to detect possible concurrency errors in different schedules based on permutation of operations in the observed trace [15, 32, 33]. They typically encode the checked property as a symbolic formula and use SMT solvers to check them. In contrast, we detect errors by leveraging the execution graph and by maintaining appropriate metadata.

Data race detection in task parallel programs. The approach proposed by Mellor-Crummey *et al.* [22] and Non-determinator [10] were seminal in proposing the detection of data races for a given input in task parallel programs using the series-parallel execution graph. Subsequently, these techniques have been enhanced to handle locks [9], to handle task graphs in Habanero-Java [27], and to detect races without serial execution with SPD3 [26, 28]. Our proposed research uses the DPST representation in SPD3 and is inspired by the access histories in the All-Sets algorithm for Cilk [9]. We extend these techniques to detect atomicity violations. Further, our metadata is not proportional to the number of dynamic accesses to a location.

There are proposals to enforce determinism in task parallel programs by detecting data races [17]. In the absence of synchronization, data race freedom ensures determinism [4, 17] which also eliminates atomicity errors. Tardis [17] checks for determinism by maintaining a log of accesses and identifying conflicting accesses between tasks. In contrast, our approach handles atomicity violations in the presence of synchronization operations.

6. Conclusion

This paper proposes a dynamic analysis to detect atomicity violations in task parallel programs. It detects errors in

the observed trace and also in different possible schedules for a given input. The key idea is to track access histories with each location and check conflict serializability of the accesses that may happen in parallel. We accomplish this by leveraging the structure of the execution, capturing a small number of access histories necessary to detect atomicity violations, and splitting the access history metadata into global and local metadata. Our approach detects atomicity errors that can occur in different schedules, provided the trace seen by our analysis contains all shared memory operations executed in other schedules in some order. Static analysis can likely be used to create an over-approximation of such a set of accesses, which we plan to explore in the future.

Acknowledgments

We thank David Menendez, Jay Lim, Aarti Gupta, John Mellor-Crummey, and the CGO reviewers for their feedback. We also thank Adrian Sampson for developing Quala. This research is supported in part by NSF CAREER Award CCF-1453086, a sub-contract of NSF Award CNS-1116682, and a NSF Award CNS-1441724.

References

- [1] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
- [2] 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.
- [3] S. Biswas, J. Huang, A. Sengupta, and M. D. Bond. DoubleChecker: Efficient Sound and Precise Atomicity Checking. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 28–39, 2014.
- [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] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 167–178, 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] Q. Chen, L. Wang, Z. Yang, and S. D. Stoller. HAVE: Detecting Atomicity Violations via Integrated Dynamic and Static Analysis. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering*, pages 425–439, 2009.
- [9] 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.
- [10] 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.
- [11] C. Flanagan and S. N. Freund. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 256–267, 2004.
- [12] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 293–303, 2008.
- [13] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive Type Qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 1–12, 2002.
- [14] 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.
- [15] 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.
- [16] D. Lea. A Java Fork/Join Framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, pages 36–43, 2000.
- [17] 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.
- [18] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: Automatically Inferring Multi-variable Access Correlations and Detecting Related Semantic and Concurrency Bugs. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 103–116, 2007.
- [19] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 37–48, 2006.
- [20] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-Aid: Detecting and Surviving Atomicity Violations. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, pages 277–288, 2008.
- [21] M. McCool, A. Robison, and J. Reinders. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann, 2012.
- [22] 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.
- [23] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pages 267–280, 2008.
- [24] S. Nagarakatte, S. Burckhardt, M. M. Martin, and M. Musuvathi. Multicore Acceleration of Priority-based Schedulers for Concurrency Bug Detection. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 543–554, 2012.
- [25] S. Park, S. Lu, and Y. Zhou. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 25–36, 2009.
- [26] R. Raman. *Dynamic Data Race Detection for Structured Parallelism*. PhD thesis, Rice University, 2012.
- [27] 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.
- [28] 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.
- [29] J. Reinders. *Intel Threading Building Blocks*. O’Reilly & Associates, Inc., 2007.
- [30] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller. Automated Type-based Analysis of Data Races and Atomicity. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 83–94, 2005.
- [31] 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.
- [32] 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.
- [33] C. Wang, S. Kundu, M. Ganai, and A. Gupta. Symbolic Predictive Analysis for Concurrent Programs. In *Proceedings of Formal Methods*, pages 256–272, 2009.