PINT: Parallel INTerval-Based Race Detector

Yifan Xu Anchengcheng Zhou Kunal Agrawal I-Ting Angelina Lee
Washington University in St. Louis
{xuyifan, ann.zhou, kunal, angelee}@wustl.edu

***Abstract*—A race detector for task-parallel code typically consists of two main components — a *reachability analysis* component that checks whether two instructions are logically in parallel and an *access history* component that keeps track of memory locations accessed by previous instructions.**

Race detectors from prior work typically utilize a hashmap to maintain the access history, which provides asymptotically optimal overhead per operation but can incur significant overhead in practice, since the detector needs to insert into and query the hashmap for every memory access. An exception is STINT by Xu et al., which race detects task-parallel code by coalescing memory accesses into *intervals*, or continuous memory locations accessed within a sequence of instructions without any parallel construct. STINT utilizes a treap to manage access history that allows for insertions and queries of non-overlapping intervals. While a treap incurs higher asymptotic overhead per operation, this strategy works well in practice as the race detector performs operation on the access history with much lower frequency compared to the strategy that utilizes a hashmap.

STINT only executes task-parallel code *sequentially*, however, due to the unique design of their treap that ensures no overlapping intervals exist in the tree. Parallelizing STINT efficiently is non-trivial, as it would require a concurrent treap that ensures no overlapping interval, which is challenging to design and likely incurs high synchronization overhead.

This work proposes PINT, a race detector that, like STINT, race detects task-parallel code at the interval granularity and utilizes the same treap design to maintain access history. PINT executes the computation in parallel, however, while keeping the parallelization / synchronization overhead low. A key insight is that, PINT separates out operations needed for race detection into the *core* part (e.g., reachability maintenance) and the *access history* part. Doing so allows PINT to parallelize the core part efficiently and perform the access history part asynchronously, thereby incurring low overhead.

I. Introduction

A particularly common and insidious cause of bugs in parallel programming is ***determinacy races*** [1] (also called ***general races*** [2]) which occur when two or more instructions which are logically in parallel with each other access the same memory and at least one of the accesses is a write. A determinacy race can cause the program to behave in a nondeterministic way depending on the order in which the operations are scheduled. Finding determinacy races is also challenging due to the nondeterministic program behavior.

For task-parallel programs, the normal paradigm is to detect races "on the fly" — as the program executes for a particular input, the race detector maintains data structures that keep track of which instruction(s) have read/written to which memory location and reports a race when conflicting accesses occur. Several algorithms have been proposed for on-the-fly race detection in task-parallel code [3], [1], [4], [5], [6], [7], [8], [9], [10], [11], [12]. These algorithms guarantee that the race detector reports a race if and only if a race exists for that input. On-the-fly race detection maintains two crucial data structures: (1) a ***reachability analysis*** data structure maintains information that allows us to determine whether two ***strands*** — a sequence of instructions containing no parallel control — are logically in parallel with each other, and (2) an ***access history*** (also called shadow memory) data structure that records (possibly a subset of) strands that have accessed a given memory location in the past. Every time a particular memory location, say ℓ, is accessed by a strand t, the race detector uses the access history data structure to determine (the appropriate subset of) prior strands that accessed ℓ. Then the reachability data structure is queried to see if any of these prior strands are logically in parallel with t. If no race is found, then t is (possibly) added to the access history (this step may involve further queries to the reachability data structure) so future strands can detect races.

Most prior work for optimizing on-the-fly race detection for task parallel programs has focused on how to update and query the reachability data structure more efficiently. The access history data structure in the prior work is generally an optimized hashmap where the key is each memory location and the value is the (list of) prior accesses to this memory location. Therefore, each query and update to the access history takes constant time. However, each memory access within the program leads to a query (and potentially an update) to this data structure. In addition, as mentioned above, each query and update to the access history also (potentially) leads to reachability data structure queries. Therefore, reducing the number of access history queries and updates can lead to significant improvement in the performance of race detection.

Researchers have observed that memory accesses within many programs exhibit spatial locality; that is, strands often access contiguous memory locations. Therefore, instead of storing individual memory locations in the access history, one can store ***intervals*** — a range of contiguous memory locations accessed by a single strand. This observation underlies work in various contexts that use compile time techniques, runtime techniques, or both to generate appropriate intervals to speed up access history management [13], [14], [15], [16], [17], [18]. Our work, in particular, is inspired by the interval-based access history STINT [18], which generates intervals using

This work was supported in part by the National Science Foundation under grant numbers CCF-1725647, CCF-1733873, CCF-1910568, and CCF-1943456.

both compile time and runtime coalescing and stores access history using a dynamic treap [19], [20] data structure. The treap by Xu et al. [18] enforces that no two intervals within the treap overlap and provides fast algorithms to insert and query into such a treap data structure. Their approach runs the input parallel program sequentially and uses one reader treap and one writer treap for access history management.

This paper describes PINT, a parallel race detector based on the treap data structure used by Xu et al. [18]. In particular, like all parallel race detection algorithms for race detecting series-parallel programs, we store (up to) three previous accessors (strands that previously accessed that memory location) for each memory location using three treaps. The primary difficulty with designing a parallel race detection algorithm that uses a dictionary for storing the access history intervals is *concurrent accesses from strands that run in parallel*. For hashmap based access history, concurrent accesses are less of a problem since each cell of the hashmap stores only one memory location and all cells can be accessed concurrently without correctness issues. When we use a treap (or any tree or skiplist data structure), two issues arise: 1. Each node stores a potentially large interval and multiple parallel strands could access different memory locations within the same interval; and 2. Since it is pointer-based balanced search-tree data structure, even accesses to different nodes may change the data structure globally in order to maintain balance conditions. This is magnified for the modified treap by Xu et al. [18] since each insert and query can traverse multiple paths in the treap.

One possible solution is to use concurrent search tree or skiplist data structures; however, we are not aware of a concurrent treap data structure that satisfies the no-overlap property we need and the complex insert and query semantics would make it nontrivial to design one. In addition, even generic concurrent search trees and skiplists often incur significant overheads due to locking or are very complicated to implement if they are nonblocking.

In this paper, we take the approach of minimizing overheads by using sequential treaps while executing the actual program in parallel. Our design separates out operations needed for race detection into the ***core*** component and the ***access history*** component, but the two components perform additional bookkeeping so that they can be parallelized independently and execute asynchronously. A set of ***core*** workers execute the core computation in parallel, managing reachability and producing coalesced access intervals that will be picked up by ***treap*** workers that manage the access history. Each treap worker manages one of the three treaps independently from each other but perform enough bookkeeping to ensure the processing of intervals follow a single global order to ensure correct race detection.

Since each treap is only accessed by a single worker, a normal sequential implementation of the treap suffices. Since the core workers and treap workers proceed asynchronously — the core workers do not wait for the treap workers to finish processing intervals for a particular strand before proceeding to the next strand — our design is compatible with any reasonable scheduling policy for the core workers. This asynchronous design enables efficiency, since the core workers are never idle waiting for treap workers to finish. Our implementation uses work-stealing to schedule the work of the core component.

Contributions. This paper makes the following contributions:

1) We propose a unique parallelization strategy to enable efficient treap-based access history for parallel race detection. An important feature of our design is *asynchronous* access history queries and updates, allowing the core workers to proceed execution without waiting for the access history updates to complete.
2) This asynchrony creates interesting issues that prior race detectors with synchronous access history does not consider. For one, to ensure correct race detection, all treap workers must process strands in one single global order and ensure that the ordering satisfies the dependences specified by the program computation. For two, recycling memory can become problematic, as the memory should not be freed until the access history has processed the corresponding interval. Our design addresses these issues, and we formally show that the proposed design provides the same correctness guarantees as prior work.
3) We have developed PINT based on the proposed design, and we experimentally evaluate PINT to assess the proposed parallelization strategy. Our experiments indicate that PINT incurs small parallelization overhead and provides decent scalability. Moreover, it executes much more efficiently in practice, providing a much better overhead compared to the state-of-the-art parallel race detector for task-parallel code.

II. Preliminaries

This section provides some necessary background.

Fork-Join Parallelism and Series-Parallel DAGs. This paper focuses on race detection for programs created using fork-join parallelism. Fork-join parallelism is typically expressed using two keywords: ***`spawn`*** — which denotes that the spawned function may execute in parallel with the continuation — and ***`sync`*** — which ensures that all locally spawned computations must return before the program proceeds.[1]

We can model a parallel computation as a ***directed-acyclic graph*** (or ***DAG*** for short), where nodes represent ***strands***, a sequence of instructions containing no parallel control (i.e., `spawn` or `sync`), and edges represent dependences. Fork-join computations generate a special class of DAGs called ***series-parallel DAGs*** [21] (or ***SP DAGs*** for short). The DAG unfolds dynamically as the program executes. When the execution encounters a `spawn`, a ***spawn node*** is created with two children. By convention, the left child is the first node of the spawned subroutine and the right child is the node representing the continuation of the parent, referred to as the ***continuation node***. When the execution successfully passes a `sync`, it creates a ***sync node*** with multiple incoming edges.

[1]The exact keywords may be different across languages and libraries, but the spirit is captured by these keywords.

Work-Stealing Scheduler. During execution, a ***work-stealing*** scheduler [22], [23] dynamically load balances a parallel computation across available ***worker*** threads. Each worker maintains a ***deque***, double-ended queue, of available work. When a worker w executes a spawn node, which creates two children, w pushes the right child (continuation node) onto the bottom of its deque and continues to execute the left child (spawned subroutine). When w returns from the spawned subroutine, it pops the bottom-most continuation node from its deque and resumes its execution. If its deque becomes empty, w turns into a ***thief***, chooses a ***victim*** worker at random to ***steal*** from, and takes the topmost node — the oldest continuation node in the deque — from the victim. Thus, on a given worker, the continuation of a spawn is always execution after the spawned function. Moreover, the execution of a worker between successful steals follows that of the sequential execution, expanding the DAG is a depth-first left-to-right fashion.

When a worker reaches a sync node, it must check if all previously spawned children have returned. If no continuation associated with any of the spawns within this sync block were ever stolen, then this is trivially true since the execution follows sequential order. We call this type of sync a ***trivial sync***, which is a nop and the worker which encounters a trivial sync can just continue to execute past it. On the other hand, if any continuation within the sync block was stolen, then it is a ***nontrivial sync*** and can only be passed once all previously spawned subroutines return. The runtime system keeps track of the number of outstanding spawned subroutines for each sync block, which allows a worker executing a nontrivial sync to determine whether the `sync` can be passed or not. If not, the worker suspends the function and go work steal next.

Race Detection in Series-Parallel DAGs. When a particular strand s accesses a memory location, say x, the on-the-fly race-detection tool checks the access history to find prior strands that have accessed this memory location in order to check if there are parallel conflicting accesses. Given a pair of nodes u and v (in this case, s and one of the relevant strands stored in the access history) the reachability component checks if they are ***in series*** — there is a path from between them (represented by $u \rightsquigarrow v$) — or ***in parallel*** — there is no path between them. If no race is detected, then we update the access history if necessary. In this paper, we use WSP-Order [7], an asymptotically optimal parallel algorithm for reachability analysis for series-parallel DAGs. We omit the detail fro WSP-Order here as it can be used as a black box in PINT.

For race detection in general DAGs, the access history data structure must store potentially many prior strands to correctly report racy programs. In particular, we need store only one writer (the last writer) per memory location, but have to store a set of readers which are all in parallel with each other. For series-parallel programs, however, it is sufficient to store a small number of prior strands in the access history. For serial race detectors (where the race-detector executes the (parallel) program sequentially), then it suffices to store one reader (and the last writer) to perform race detection correctly in series-parallel programs [1].

More relevant to this paper, for parallel race-detectors, it is sufficient to store two readers and one writer (still the last writer) per memory location [3]. The two readers stored are the ***left-most reader*** and ***right-most reader***. The left-most reader is the one that would execute first *among all parallel readers* in a left-to-right depth-first execution of the DAG and the right-most reader is the one that would execute last. Another, informal way of thinking about this is that the left-most (correspondingly, the right-most) reader appears left-most among the parallel readers to the memory location in the conventional drawing of the DAG. It is important to note the phrase "among the parallel readers" — for instance, if u and v are the left-most and the right-most readers to x, and a node w, which is sequentially after both of them reads x subsequently, then w will replace u and v as both the left-most and the right-most readers. The structural properties of series-parallel DAGs ensure that if both the left-most and the right-most readers are in series with some subsequent strand, then all other parallel readers are also in series with it.

III. PINT Implementation Overview

This section overviews the implementation of our race detector, PINT, and discusses its design rationale.

The design choices in PINT align with two specific goals. First, PINT should provide the traditional correctness guarantee of on-the-fly race detection. Specifically, PINT reports a race if and only if a given input program contains a race. Second, PINT should provide efficient practical performance. To achieve this goal, the design of PINT strives to incur low parallelization overhead, avoid synchronization whenever possible, and utilize a load-balancing strategy that leads to balanced workload across available cores. In this context, we define ***parallelization overhead*** as the additional work that PINT must do in order to enable parallel execution.

A. Race-detection with an Interval-Based Access History

Our work is a parallelization of STINT [18], a sequential race detector with interval-based access history. STINT uses two interval treaps to perform race detection — the writer treap holds the last writer for each interval and the reader treap holds the relevant reader for each interval. In STINT, compile time and runtime coalescing is used at the strand level — at the end of a strand s, a set of read and write intervals are generated which contain all the memory accesses by s. At this point, the read intervals of s are checked against the writer treap and then inserted into the reader treap. Similarly, the write intervals are checked against the intervals in both treaps and then inserted into the writer treap. The two main differences between STINT and traditional race detection are: 1. interval-based access history; and 2. delayed processing of accesses since the intervals of a strand are only checked against and inserted into the access history once the strand finishes executing.

One important thing to note is that the treaps store the exact intervals and not approximate intervals. This is facilitated

by ensuring that no two intervals in the treap can overlap — the insertion process ensures that each memory location is in at most one interval and this interval is the "correct interval." For instance, say the writer treap has intervals $[1, 4, u]$ and $[6, 10, v]$ (indicating that the last writer to memory locations from 1 to 4 was strand u and the last writer to memory locations from 6 to 10 was v. If a strand w now writes to interval $[3, 7]$, the treap will be modified to contain intervals $[1, 2, u], [3, 7, w], [8, 10, v]$. The reader treap is even more complex, but based on similar principles. This allows STINT to provide the precise guarantee we want — a race is detected if and only if there is a race in the program.

Our system, PINT, uses the same compile time and runtime coalescing as well as the same treap data structure as PINT. In order to provide the same correctness guarantee in the presence of parallelism, instead of maintaining two treaps, we must maintain three treaps — writer treap, left-most reader treap, and right-most reader treap. The insertion process for the last writer and the left-most reader treaps are identical to STINT while the insertion process for the right-most reader treap is complementary to the left-most reader treap. Our main contribution is design of mechanisms that allow for efficient parallelization as described below.

B. Challenges in Efficient Parallelization

PINT, like STINT, maintains access history with an interval-based treap. As a strand executes, it generates intervals using compile time and runtime coalescing; these intervals are inserted into treaps for correct race checking. The primary challenge with efficient race detection is *concurrency* — since multiple logically parallel strands can execute and generate memory accesses in parallel, a treap must coordinate the concurrent updates and queries from these strands.

One way to handle this is to design a concurrent treap which allows multiple queries and inserts to run at the same time. However, concurrent tree-like data structures are often difficult to implement and inefficient. In addition, since the treap must satisfy the property that no two intervals overlap, we cannot simply use any existing design of concurrent binary search tree. Moreover, the non-overlapping property requires each insert or query operation to potentially traverse multiple paths in the treap [18], likely leading to high synchronization overhead for concurrent operations.

C. Overview of Our Parallelization Approach

In order to execute the computation in parallel and race detect without incurring high overhead, PINT explicitly separates out the access history management from the rest of the operations required by race detection. To maximize available parallelism, PINT exploits available parallelism within each component and allows them to execute *asynchronously*.

We refer to operations involving access history as the ***access history*** component, which includes inserting intervals into the treap data structures and performing queries on them to check for races. PINT maintains three treaps: one treap each for last writer, left-most reader and right-most reader. We refer to the rest of the operations as the ***core*** component, which includes executing the core computation, maintaining reachability, and coalescing memory accesses within a strand into intervals.

This separation allows us to easily exploit parallelism inherent in the core component with WSP-Order, the same algorithm used by prior work C-RACER [7], which only incurs constant overhead. Each treap data structure in the access history component is a sequential treap and they can be managed *almost* independently. The only coordination required is that each treap must process the same sequence of reads and writes. When PINT executes on P processing cores, it utilizes three cores for the access history component, one for each treap, and allocates $P - 3$ cores to the core component.

D. Minimizing Synchronization in Asynchronous Execution

Our parallelization strategy avoids any parallelization overhead within treaps, but requires some coordination between the two components to allow for asynchronous execution and between the three treaps so that all treaps observe the same access history. We refer to workers executing the core component as ***core workers***, and workers performing operations on each of the treaps as the ***left-most reader treap worker***, ***right-most reader treap worker***, and ***writer treap worker***.

To enable asynchronous execution, each core worker, as it executes, deposits each executed strand with its interval information into a first-in first-out ***trace*** data structure. Each core worker has its own local trace data structure, such that they can execute strands independently from each other and ahead of the treap workers, processing the core computation without waiting for the access history component to process the executed strands.

The treap workers process the generated intervals in a single global order. Specifically, PINT dedicates one treap worker, the writer treap worker, to collect strands from the trace data structures of each core worker and move them into a (first-in first-out) ***access history queue*** shared among treap workers. Each treap worker processes strands in the order specified by the queue, which allows each treap worker to process strands independently but still maintains the same global ordering of access history.

The goal of this strategy is to minimize coordination and synchronization. The only coordination between components is on the trace data structure (which is a producer-consumer queue) where each core worker produces intervals corresponding to each strand it processes on its (local) trace data structure and the write treap worker subsequently collects these intervals. This requires synchronization only when the queue contains one or zero elements. The synchronization on the access history queue is also minimal since only the writer treap worker modifies it and the reader treap workers only read it. Small amount of synchronization is necessary to allow for proper recycling of strands and slots in the queue — this is implemented using simple atomic fetch-and-add counters to keep track of how many treap workers have finished processing a strand. The writer treap worker can free memory allocated for the strand and reuse the slot once the counter reaches three.

Finally, this strategy provides decent load balancing, as we demonstrate experimentally in Section IV. The work due to the access history component is generally much smaller than the core component, at least in our tested benchmarks. Therefore, with three-way parallelization, the access history component is not the bottleneck when running on 20 (or fewer) cores. In addition, executions generally have more reads than writes, making the writer treap worker the least busy; this motivated our decision to dedicate writer treap worker to collect the executed strands, update and recycle the access history queue.

E. Conforming to DAG Dependences

To perform parallel race detection correctly, each treap worker must process strands in an order that satisfies the dependences specified in the computation DAG (defined in Section II). That is, if strand $u \rightsquigarrow v$, then the treap must process the accesses of u before accesses of v. If u and v are in parallel, their accesses can be processed in any order. Henceforth, we will refer to an order that satisfies these requirements as ***DAG-conforming***. We now describe how PINT ensures that if $u \rightsquigarrow v$, then u will be before v in the access-history queue as well.

Operations on the Trace Data Structures. Algorithm 1 shows the operations that a core worker performs to populate the trace data structures that contain executed strands. We first focus on how a core worker populates a trace data structure, discuss the properties of a trace data structure, and finally examine how PINT ensures that strands are processed in a DAG-conforming order.

A core worker inserts each executed strand into its current trace data structure according to their execution order (lines 11, 18, 25, 32, and 38). Whenever the core worker executes a stolen continuation or a non-trivial `sync`, it puts away its current trace and starts a new trace (lines 22–24 and lines 35–37). A core worker additionally performs the necessary bookkeeping with executed strands to allow the writer treap worker to check whether a strand v is ready to be collected.

Properties of Trace Data Structures. Before we discuss the bookkeeping necessary to enable the writer treap worker to collect strands in a DAG-conforming order, we first formally define the properties that trace data structure maintains.

Lemma 1 (Trace Properties): A trace data structure satisfies the following properties:

1. The ordering of strands within a single trace data structure follows that of the sequential execution;
2. If a strand v is the first strand in a trace $\mathcal{T}$, then v must be either a stolen continuation of a spawn node or a sync node corresponding to a non-trivial `sync`, and v's immediate predecessor(s) must be in a different trace.
3. If a strand v is not the first strand in a trace $\mathcal{T}$, v's immediate predecessor(s) must also be in $\mathcal{T}$.

Proof. Property 1 trivially holds because a core worker's execution follows that of the sequential execution between successful steals and a core worker inserts strands into a trace data structure based on their execution order.

The first part of Property 2 holds simply by construction — based on Algorithm 1, a core worker obtains a new trace only when it encounters a stolen continuation (lines 22–24) or a non-trivial `sync` (lines 35–37). The second part of Property 2 holds because any of v's immediate predecessor u must execute before v and therefore u can not appear after v in $\mathcal{T}$ due to Property 1.

Finally, we argue that Property 3 holds. Since v is not the first strand in $\mathcal{T}$, v is neither a stolen continuation nor a non-trivial sync node. If v has one immediate predecessor u, Property 3 trivially holds because there is nothing between the execution of u and v to cause the worker to switch to a new trace. If v has multiple immediate predecessors, v must be a trivial sync node; that is, no continuation within the sync block was stolen. In this case, v's immediate predecessors, which include the return nodes of these subroutines and the code leading up to the `sync` statement corresponding to v, must execute on the same core worker without intervening steals and be in trace $\mathcal{T}$ also. □

Algorithm 1: Trace operations done by core workers

```
1  let w be the executing worker
2  let u be the node representing currently executing strand
3  /* invoked upon executing a spawn */
4  Function Spawn(u) // u is the spawn node
5      let c be the continuation node and s be the sync node
6      u.child ← c
7      u.sync ← s
8      c.pred ← 1
9      if u is the first spawn for s then
10         s.pred ← 0
11     w.trace.Insert(u)
12 /* invoked upon executing a return from a spawned function */
13 Function SpawnReturn(u) // u is the return node
14     let s be the corresponding spawn node
15     if s.child is stolen then
16         u.child ← s.sync
17         s.sync.pred ← s.sync.pred + 1
18     w.trace.Insert(u)
19 /* invoked upon executing the continuation of a spawn */
20 Function Continuation(u) // u is the continuation node
21     let s be the corresponding spawn node
22     if u is stolen then
23         w.tracepool.Put(w.trace)
24         w.trace ← new Trace()
25     w.trace.Insert(u)
26 /* invoked upon encountering a sync */
27 Function Sync(u) // u is the strand leading to the sync
28     let s be corresponding sync node
29     if s is a non-trivial sync then
30         u.child ← s
31         s.pred ← s.pred + 1
32     w.trace.Insert(u)
33 /* invoked after passing a sync successfully */
34 Function AfterSync(u) // u is the sync node
35     if u is a non-trivial sync then
36         w.tracepool.Put(w.trace)
37         w.trace ← new Trace()
38     w.trace.Insert(u)
```

Strand Processing Order. The writer treap collects and processes strands one by one. We say that a strand is ***ready*** to be collected once its immediate predecessor(s) have been collected. The writer treap ***collects*** a strand by finding a ready

strand s from a core worker and inserting s into the access history queue; then, it ***processes*** s by checking races with s's read intervals against the writer treap and inserting s's write intervals into the writer treap. The reader treap workers process the strands in the same order as the writer treap worker by following the order in the access history queue and perform complementary checks and insertions. Each treap worker can process strands independently as each operates exclusively on its own treap.

Since all treap workers process the strands based on their order in the access history queue (first-in first-out), if the collection order is DAG-conforming, so is the processing of the strands. The writer treap worker follows two simple ***collection rules*** when collecting strands from core workers:

1. it collects strands from a trace $\mathcal{T}$ only if the first strand v in $\mathcal{T}$ is ready; and
2. it collects strands from a trace $\mathcal{T}$ in first-in first-out order.

Lemma 2: (Collection Rules) The collection rules ensure that the writer treap worker collects the strands in a DAG-conforming order.

Proof. For a strand v that is $\mathcal{T}$'s first strand, by Rule 1 v can only be collected after its immediate predecessor(s) are collected. For a strand v that is not the first strand in $\mathcal{T}$, we know that any of v's immediate predecessor(s) u is also in $\mathcal{T}$ by Property 3 in Lemma 1 and u must be before v in $\mathcal{T}$ by Property 1 Lemma 1. Thus by Rule 2, v must be collected after its immediate predecessor(s) are collected. These two rules together suffice to ensure that the collection of strands follows a DAG-conforming order. □

Algorithm 2: Strand collection done by the writer treap worker

```
39 let u be the strand that the writer treap worker is collecting
40 Function Collect(u)
41 |   queue.Insert(u)
42 |   if u is a spawn node or
43 |     u.child is set and u.child is a non-trivial sync then
44 |       u.child.pred = u.child.pred − 1;
```

Checking for Strand Readiness. Rule 2 is not difficult to implement as a trace data structure is first-in first-out. Rule 1 in Lemma 2 requires the writer treap worker to check whether the first strand v in a trace $\mathcal{T}$ is ready. By Property 2 in Lemma 1, we know v is either a stolen continuation of a spawn node or a sync node that corresponds to a non-trivial `sync`. Algorithm 1 shows how a core worker maintains a count with such a strand v the number of its immediate predecessor(s) (denoted as $v.pred$) and a pointer from its immediate predecessor u back to v (denoted as $u.child$) to enable the check. Algorithm 2 shows the `Collect` operation for the writer treap worker — when it collects a strand u, it decrements the count of u's immediate successor (denoted as $u.child.pred$) when appropriate. The writer treap worker invokes `Collect` on a strand u only when its count reaches zero, indicating that u is ready to be collected.

Lemma 3 (Strand Readiness): For the first strand v in a trace, v is ***ready*** to be collected, (i.e., all its immediate predecessor(s) has been collected), if v's count reaches zero.

Proof. By Property 2 in Lemma 1, v is either a stolen continuation node or a sync node of a non-trivial `sync`.

If v is a continuation node, its $v.pred$ is always set to one (line 8) and decremented when its predecessor (the corresponding spawn node) is collected (lines 42 and 44), regardless of whether the continuation is stolen or not. Thus, v must be ready if its count is zero. As a side note, even though the count is not checked when v is not the first node in the trace, we chose to set and decrement its count always to simplify the implementation.

If v is a sync node, its $v.pred$ is initialized upon encountering the first `spawn` within the sync scope (lines 9–10). A sync node v's immediate predecessors include the return node (last strand) of all the spawned functions within its sync scope and the continuation of the last `spawn` statement in its sync scope leading to the `sync` statement. When a core worker executes one of a sync node v's immediate predecessors u, if v corresponds to a non-trivial `sync` (which is known when u executes), the core worker sets u's child pointer to v and increment v's count (lines 15–17 and lines 29–31). For each such immediate predecessor u of v that performed the increment, there is a corresponding decrement when the writer treap invokes `Collect(`u`)` shown in Algorithm 2, as `Collect` checks if u has its child set to a non-trivial `sync` and decrement the child's count if so (lines 43 and 44). Thus, if v's count reaches zero, all its immediate predecessor(s) have been collected and v must be ready. □

Lemma 4 (DAG-Conforming Order): All treap workers process strands in a DAG-conforming order.

Proof. By Lemmas 1, 2, and 3, we know that the writer treap worker collects strands in a DAG-conforming order. The statement follows since all treap workers follow the same order to process strands via the access history queue. □

F. Avoiding Address Translation

The strands within a trace follow the sequential execution order (by Property 1 in Lemma 1). The sequential execution order is DAG-conforming but stricter than what we need. In principle, two nodes which are logically in parallel within the same trace can be collected in any order. We specifically chose to collect nodes in trace order since it has two important performance advantages. First, a core worker's execution between successful steals follows the sequential execution order and is therefore naturally DAG-conforming removing need for some bookkeeping. Additionally, it allows us to implement the trace data structure to be a simple resizable array.[2] Second, and more importantly, processing strands in a given trace data structure in the sequential order allows us to manage access history in a simpler manner than would otherwise be possible due to stack reuse.

[2]We have used a linked list of fixed-sized array in our implementation.

To understand the need for address translation, consider a scenario where a function A spawns function B and subsequently calls function C in the continuation after spawning B. Assuming the continuation that calls C is not stolen and thus executed on the same worker, B and C will share the same stack space for their respective execution potentially in a conflicting way. Without any special handling, a race detector will report a race, since B and C are logically in parallel, but this would be a false race, since their stack frames are logically distinct and share the same addresses only due to stack reuse.[3]

A race detector typically handles this scenario by clearing the range corresponding to B's stack frame from the access history once B returns before executing C that reuses the space. In PINT, this simple strategy doesn't quite work because the access history component executes *asynchronously* — when a core worker finishes executing B and goes on to execute C, the intervals accessed by strands in B (and its sub-computation) may not have been fully processed by the treap workers yet. Nevertheless, since PINT processes strands within a single trace following sequential execution order, we know that the return node of B will be processed before any strands of C. Thus, to handle this scenario in PINT, each treap worker simply clears out the memory range of B's stack frame from its own treap once it processes the return node of B. If, on the other hand, we had allowed B and C to process out of order even though they are in the same trace, then, without special handling, we might detect these false races.

The asynchronous access history component poses a similar but slightly different issue for heap memory reuse. Again imagine the same code example where A spawns B and calls C. Imagine B frees some heap memory and C subsequently allocates heap memory, which happens to be reusing the memory freed by B. Also note that this scenario can happen even if B and C execute in parallel on two different core workers (and thus in two different traces). Again, a race detector with synchronous access history can simply clear out the access history of the freed memory range upon a call to `free`. PINT cannot do that due to asynchronous access history — when a core worker executes the `free`, strands prior to `free` that use the memory may not have been fully processed yet. To handle this issue, PINT simply delays the actual `free` call — when a core worker executes a `free` call in strand s, the "freed" memory is put aside being stored with the strand s inserted into the trace. When the writer treap collects and processes s, it performs the actual `free` to free the memory. It is safe to perform the actual free once s is collected because any strand that reuses the freed memory later must be collected after s.

G. Putting Everything Together

We have discussed the parallelization strategy, the asynchronous design, and the properties that the access history guarantees. In the last part of this section, we argue that our design provides the desired correctness guarantee, i.e., PINT reports a race if and only if a race exists in the computation. Recall that PINT detects races at the strand granularity where the queries and updates to the access history are delayed and asynchronous. On the other hand, other race detection algorithms such as C-RACER [7] generally perform queries and updates to the access history as soon as a memory access occurs. Therefore, one might worry that PINT does not provide the same correctness guarantee. Here we will prove that it, in fact, does.

Theorem 5: Say the treap workers in PINT process strands in some order $\mathcal{H}$ based on the access history queue, and a different parallel race detector, say A, executes the strands one at a time in the same order $\mathcal{H}$ but uses a synchronous access history component. Then, PINT reports a race between strands u and v if and only if A reports a race between the same strands u and v.

Proof. By Lemma 4, we know that $\mathcal{H}$ is a valid schedule for A. Consider the two executions — 1) execution of PINT with the access history queue containing strands in $\mathcal{H}$, and 2) execution of A executing strands in $\mathcal{H}$ one at a time but insert and queries the access history as each memory access occurs. By induction on $\mathcal{H}$, before PINT processes a strand s, the state of each memory location in the access history (i.e., left-most reader, right-most reader, and last writer) is the same as that in A before A executes s.

If A does not detect a race, it's obvious that PINT will not, either, as there are no conflicting accesses in the access history. Say A detects a race between strands u and v. That is, when A executes v, it finds a conflicting access u, where u is not v (as accesses in v cannot race with each other). The same strand u is also in PINT's access history before it processes v. In PINT, since a treap worker always performs query first before insert when it processes a strand, PINT will also detect a race between u and v. □

Note that we do not claim that PINT and A will report exactly the same races. Rather, they simply report a race between the same pair of strands. For instance, if strand u reads memory location x and is stored in the access history as the left-most reader of x (and some other strand w is stored as the right-most reader to x). Later a strand v (which is in parallel to u and left-of u) reads x and then writes x, A will not detect the write/read race between u and v since v will replace u as the left-most reader to x when it reads x and the subsequent write to x by v will not be detected as the race. A will instead detect the write/write race between u and v (and also detect a different write/read race between w and v). On the other hand, PINT will detect all three races if let to run long enough.

IV. Experimental Evaluation of PINT

In this section, we empirically evaluate PINT to gauge whether the design choices we made achieve the desired goal, namely practical parallel performance. Specifically, we would like to answer the following questions:

1. how the performance of PINT compare to prior work;

[3]This scenario does not cause an issue when B and C execute in parallel, because they will not share stack space thanks to the runtime support for "cactus stack" [24].

2. how much parallelization overhead PINT incurs; and
3. how well can PINT scale and under what condition the use of sequential treaps may create a bottleneck.

Experimental Setup. We ran our experiments on a machine with two Intel Xeon Gold 6148 processors, each with 20 2.40-GHz cores. Each core has a 32-KB L1 data cache, a 32-KB l1 instruction cache, and a 1-MB L2 cache. Each processor has a shared (among 20 cores) 27.5-MB L3 cache and the whole system has 768 GB main memory. Hyperthreading is disabled. Experiments ran in Linux kernel 4.15, and all benchmarks were compiled with -O3 using the clang 10.1-based Tapir compiler [25].

To evaluate the empirical performance, we compare PINT with two other race detectors from prior work. The first one is STINT [18] with a similar design for access history, although STINT must execute the computation *sequentially* and has a synchronous access history. As far as we know, STINT is the only race detector for task-parallel code that focuses on optimizing the access history component. The second system is C-RACER [7], which implements the state-of-the-art WSP-Order algorithm for reachability analysis. We compare PINT against C-RACER because PINT utilizes the same WSP-Order algorithm for reachability and mainly differs in how it handles access history.

We evaluate and compare these systems using seven task-parallel benchmarks, including Cholesky decomposition (`chol`, $n = 4000$, $z = 40000$, $b = 16$); parallel mergesort (`sort`, $n = 5e^7, b = 2048$); fast Fourier transform (`fft`, $n = 2^{26}, b = 128$); heat diffusion simulation on a 2D grid (`heat`, $nx = 2048, ny = 2048, b = 10$); matrix multiplication (`mmul`, $n = 2048, b = 64$), and two versions of Strassen's algorithm for matrix multiplication, `stra` and `straz`, which use row-major order layout and Morton Z layout, respectively ($n = 2048, b = 64$). Each data point presented is the average of five runs, with standard deviation less than 5%. We collected the numbers shown in Sections IV-A and IV-B using base case size b and other benchmark-specific input sizes mentioned above, with the parallel running times collected on a single socket with 20 cores to avoid cross-socket overhead. We vary the number of cores used and input sizes when we examine the scalability and potential bottleneck of PINT in Section IV-C.

A. Overview of Performance Results

Figure 1 shows the execution times of the baseline (i.e., running the benchmark with no race detection) and on each race detector running on one core and 20 cores. When we run PINT on one core, the access history of PINT does not execute asynchronously. Rather, PINT performs the core component entirely, before moving onto the access history component (which may result in a larger memory footprint than if it had executed the access history component asynchronously on a separate thread). By removing the asynchronous maintenance of access history, the one-core execution time reflects the total amount of work involved in PINT and allows us to gauge and compare the overhead of PINT against STINT and C-RACER more easily.

This comparison allows us to gauge parallelization overhead of PINT, because both STINT and PINT perform memory coalescing using the same mechanism, race detect at the interval granularity, and utilize similar access history design. The one-core execution removes the asynchrony factor, allowing the running time to reflect the total amount of work involved in using STINT and PINT.

We first compare the execution times of PINT and STINT, both of which perform memory coalescing using the same mechanism, race detect at the interval granularity, and utilize the same treap data structure for maintaining access history. PINT performs more work than STINT in the core component and maintains one extra treap to enable parallel race detection, however. As expected, PINT incurs slightly higher overhead compared to STINT when running on one core. Nevertheless, the additional overhead allows PINT to run the computation in parallel during race detection, and the higher overhead is easily compensated by the speedup obtained during parallel execution.

We also compare the execution times of PINT and C-RACER. The numbers indicate that C-RACER incurs much higher overhead when running on one core. This is because C-RACER uses the conventional approach for maintaining access history, which detects races at each memory access. Each memory access essentially translates to multiple function calls (invoking the instrumentation hook into the race detector, which in turn invokes the insert and query into the access history data structure, which in turn invokes the query into the reachability data structure). On the other hand, the race detection overhead in STINT and PINT is more manageable compared to C-RACER, because the access history is only updated and queried much less frequently, at each interval. The reachability component in STINT and PINT is also queries much less frequently, which is determined by how many existing intervals overlap with the newly inserted interval, which is typically much smaller than the number of memory accesses represented by the intervals.

The only exception is `fft`, where C-RACER outperforms PINT and STINT. As explained in [18], this is because even though the number of intervals is smaller than the number of 4-byte memory accesses in `fft`, the reduction in the number of function calls is not sufficient to offset the increased overhead per insert and query into the treap data structure. Since PINT follows the same design of access history in STINT, PINT suffers from the same issue.

Next, we compare the parallel execution times of PINT and C-RACER shown on the right-hand side of the table. STINT is not included as it is a sequential tool. PINT maintains the same lower-overhead, although in a few benchmarks, it scales less well compared to C-RACER. However, the much lower overhead means that PINT still wins out significantly in terms of the absolute execution time.

B. Parallelization Overhead of PINT

We compare PINT against STINT when running on one core to gauge its ***parallelization overhead***, defined as the extra work

	running time on one core				*running time on 20 cores*		
	baseline	*STINT*	*PINT*	*C-RACER*	*baseline*	*PINT*	*C-RACER*
chol	4.82	154.12 [31.98×]	207.96 [43.15×]	1122.54 [232.89×]	0.28 (17.21×)	16.63 (12.51×)	58.16 (19.30×)
heat	4.09	25.68 [6.28×]	26.43 [6.46×]	599.67 [146.62×]	0.39 (10.49×)	2.21 (11.96×)	34.08 (17.60×)
mmul	8.17	230.67 [28.23×]	243.83 [29.84×]	426.69 [52.23×]	0.41 (19.93×)	13.32 (18.31×)	21.58 (19.77×)
sort	7.33	35.68 [4.87×]	50.36 [6.87×]	215.72 [29.43×]	0.37 (19.81×)	3.29 (15.31×)	29.39 (7.34×)
stra	1.64	38.58 [23.52×]	43.14 [26.30×]	594.63 [362.58×]	0.20 (8.20×)	3.09 (13.96×)	56.31 (10.56×)
straz	1.56	57.87 [37.10×]	77.14 [49.45×]	336.05 [215.42×]	0.11 (14.18×)	6.42 (12.02×)	27.67 (12.14×)
fft	13.88	549.71 [39.60×]	741.65 [53.43×]	466.31 [33.60×]	1.07 (12.97×)	276.73 (2.68×)	71.91 (6.48×)

Fig. 1. Running time of seven benchmarks shown in seconds. The left-hand side of the table shows the single-core running time of the baseline (i.e., no race detection), STINT, PINT, and C-RACER. The numbers in the brackets show the race detection overhead of each system compared to the baseline. The right-hand side shows the 20-core running time of the baseline, PINT, and C-RACER. The numbers in the parentheses show the scalability compared to its respective single-core execution.

	par. overhead	*work breakdown*				*par. exe. time*	
		core	*writer*	*rreader*	*lreader*	*core*	*total*
chol	1.35	192.18	5.14	5.37	5.27	16.32	16.63
heat	1.03	24.25	0.61	0.91	0.66	2.16	2.21
mmul	1.06	216.09	6.53	10.38	10.83	13.04	13.32
sort	1.41	45.16	0.81	2.25	2.14	3.13	3.29
stra	1.12	40.86	0.45	1.18	0.65	3.09	3.09
straz	1.33	71.61	1.76	1.81	1.96	6.28	6.42
fft	1.35	175.78	179.74	190.79	195.34	21.04	276.73

Fig. 2. The left-hand side of the table shows the parallelization overhead and the work breakdown of PINT, where the parallelization overhead is computed by taking the one-core execution time of PINT divided by that of STINT, and the work breakdown shows how much time PINT spends on the core component (*core*), the writer treap (*writer*), the right-most reader treap (*rreader*), and the left-most reader treap (*lreader*) during one-core execution. The rigth-hand side of the table shows the parallel executime time of PINT running on 20 cores spent on the core component (*core*, which uses 17 workers) and in total (*total*.

that PINT needs to perform to enable parallel execution. This overhead includes building the trace data structures in the core component, moving strands from the trace data structures into the access history queue, and managing one additional reader treap as required by parallel race detections.

As shown in Figure 2, the parallel overhead is manageable, with at most 41% increase (`sort`). To figure out where the time went, we separately collected data on how much work (computation time spent running on single core) going into each of the PINT components. The *core* component includes maintaining the reachability analysis and the trace data structures. The *writer* component involves work done by the writer treap, which includes moving strands from the data structure into the global access history queue and maintaining the writer treap. The *rreader* and *lreader* involves work done by the right-most and left-most reader treap workers, respectively, i.e., maintaining the two reader treaps.

The numbers indicate that the work done by any treap worker is relatively small compared to the work done by the core component at least for the default inputs, with the exception of `fft`. Given this breakdown, we can deduce that most parallelization overhead of PINT comes from the core component, which does the extra work of building the trace data structures. In STINT, the time spent maintaining the writer and reader treap is similar to that in PINT, with the writer treap incurs slightly less work because STINT does not need to move strands from the trace data structures into the global access history queue. However, in all benchmarks, it is more expensive to maintain the reader treaps than the writer treap, which is expected, because most benchmarks perform many more reads than writes. Thus, the writer treap is not going to be the bottleneck for the treap component.

C. Scalability of PINT

One potential concern with our design is that the use of sequential treaps for access history, albeit the three-way parallelism, would cause a sequential bottleneck during parallel execution, thereby limiting the scalability of PINT. For the benchmarks tested with default inputs running on 20 cores at least, the use of the sequential treaps is not usually the bottleneck, with the exception of `fft`, which we explained earlier. We can see this by examining the right-hand side of the table in Figure 2.

The first column in Figure 2 shows the absolute running time of the core component on 17 cores. The second column shows the overall 20-core execution time of PINT. The times shown in two columns have little difference (except for `fft`), meaning that the asynchronous access history component works well and overlap with the core component substantially during execution. Moreover, the time that each treap worker spent (which is just the total work shown on the left) is less than the parallel running time spent on the core component, which suggests that there is more room for the treap workers before they become the bottleneck.

If we had used more cores to run PINT, would the treap workers become the bottleneck? Whether the use of sequential treaps causes a bottleneck depends on a couple factors — the amount of parallelism in the core part of the computation and the total work for each treap, both of which are application and input dependent. We examine the question of scalability in more detail next and discuss when the use of sequential treaps becomes a bottleneck.

Strong Scalability Analysis. Assuming the core component scales well, the treap component may eventually become the bottleneck when running on more cores. We performed a strong scalability analysis and see how the benchmarks with the default inputs scale as we increase the core counts.

Figure 3 shows the execution times of four benchmarks when running with 1, 4, 8, 16, 24, and 32 core workers (plus three treap workers). Due to space constraint, we don't show the numbers for `chol` and `straz`, which have a similar

	# core workers used					
	1	*4*	*8*	*16*	*24*	*32*
heat	24.82	6.59	3.68	2.37	2.10	2.23 (2.04)
mmul	216.86	54.36	27.33	13.91	16.17 (9.74)	16.52 (7.38)
sort	45.75	11.79	6.14	3.39	4.06 (2.89)	4.24 (2.65)
stra	41.15	10.67	5.64	3.22	2.53	2.23

Fig. 3. Running time of four benchmarks shown in seconds running with varying number of core workers. For a given entry, the number in a parenthesis (when shown) indicates the corresponding parallel running time of the core component, where the overall running time is dominated by the treap component. Otherwise, for an entry without a parenthesis, the core component dominates the overall running time.

scaling pattern as `heat` and `mmul`, respectively; we also don't show the numbers for `fft`, as it simply does not scale.

The treap component is not the bottleneck when running with 16 core workers or less. When running with 24 and 32 core workers, the core component continues to scale (albeit slowly flattening out), and the treap component becomes the dominating overhead in some benchmarks. It's also worth noting that the race detection runs on two sockets, and the treap workers may perform cross-socket communication when they process strands from the core workers, which causes ***work inflation***, where the time it takes for a treap worker to process the same set of strands increases when we increase the core count. As an evidence supporting this hypothesis, for the two-socket runs, pinning the treap workers on the full socket (i.e., all 20 cores are used) provides slightly better performance than pinning them on the other socket with fewer core workers. Nevertheless, the performance should not degrade too much further as we increase the core counts, because the scalability of the core component is tapering out, and the amount of the treap work stays fixed sans work inflation.

Weak Scalability Analysis. We additionally perform weak scalability analysis. Typically, the weak scalability analysis involves increasing the problem size and the core count such that the work from the parallel part of the computation per core stays the same. Such an analysis is meant to demonstrate whether the overhead per unit of work stays the same as one scales out the problem size. In task-parallel code, it is difficult to measure the overhead per unit of work directly, however, because scheduling occurs dynamically during program execution. Moreover, in the context of race detection, how well the execution scales depends on two factors: the inherent parallelism in the baseline computation (which changes with the problem size) and where the race detection incurs overhead. The parallelism profile of the race detection run can differ from that of the baseline, for instance, if the race detection overhead is unevenly distributed across the computation such that higher overhead is incurred along the ***span***, the longest sequential dependences in the computation. In performing the weak analysis, we would like to tease out the impact on scalability due to where the race detector incurs overhead from the changes in the inherent parallelism of the baseline, as we care about the former but not the latter.

Thus, to perform weak scalability analysis, we increase the problem size and the core count, and we compare the running times of PINT against that of the baseline program. Assuming that scaling out does not adversely impact the scalability of the race detector, the race detection overhead (i.e., running times of PINT over that of baseline) should remain the same.

		# core workers used					
		1	*2*	*4*	*8*	*16*	*32*
heat	*baseline*	0.42	0.43	0.49	0.58	1.02	1.91
	PINT	3.07	3.19	3.42	3.68	3.95	7.38 (5.31)
	overhead	7.31	7.42	6.98	6.34	3.87	3.86
mmul	*baseline*	0.11	0.20	0.25	0.43	0.51	1.11
	PINT	3.40	6.09	6.87	12.51	13.91	132.57 (29.13)
	overhead	30.91	30.45	27.48	29.09	27.27	119.43
sort	*baseline*	3.48	3.86	3.91	4.09	4.18	5.81
	PINT	21.44	23.28	23.90	25.86	29.94 (27.51)	187.77 (41.83)
	overhead	6.16	6.03	6.11	6.32	7.16	32.32
stra	*baseline*	0.01	0.02	0.08	0.31	1.29	7.97
	PINT	21.44	23.28	23.90	25.86	29.94	90.68 (90.43)
	overhead	15.00	24.00	20.13	18.19	16.93	11.38

Fig. 4. The first two rows of a given benchmark show the running times (in seconds) of its *baseline* (no race detection) and running on PINT. The last row shows the correponding overhead (PINT/ baseline). The *baseline* is run on P cores where P equals to the number of core workers used in PINT so that it scales similarly as PINT. For a given entry, the number in a parenthesis (when shown) indicates the corresponding parallel running time of the core component, where the overall running time is dominated by the treap component. Otherwise, for an entry without a parenthesis, the core component dominates the overall running time.

Figure 4 shows the results for the same subset of benchmarks. The problem size of `heat` and `sort` doubles as we double the number of core workers. For `mmul` the input matrices' dimension is scaled up by $1.5\times$ as we double the number of core workers. For `stra` the input matrices' dimension is doubled as we double the number of core workers. Doing so gives us a varying range of how the parallelism of the computation grows as we double the core workers.

For `heat` and `stra`, the overhead of PINT decreases as we increase the problem size. Even though for the 32-core data point, the running time is dominated by the treap component, the difference between the core and the treap is small enough that the overhead is not adversely impacted.

For `mmul` and `sort`, the overhead of PINT stays about the same except for the last data point running on 32 cores. In this configuration, the running time is dominated by the treap component, which incurs a much longer running time compared to the core component, due to two reasons. First, the core component scales well in these benchmarks running on 32 cores. Second, the problem size used for the 32-core runs in these benchmarks is large enough that the treap component has quite a bit more work to do, since the increase of problem size also increases the number of intervals. It's worth noting, however, despite the treap component being the bottlenecks, PINT still perform much better than C-RACER for these benchmarks with these larger inputs. Nevertheless, these results indicate that the treap component may benefit from further parallelization for certain benchmarks with larger inputs.

V. Related Work

Access History. Access history is used in many tools such

as memory checkers and race detectors [26], [27], [28], [29], [30], [31], [1], [6], [32], [33], [7]. Many schemes have been explored to optimize access history [34], [29], [30], [28], [35], [36], [37], [38] which make various trade-offs in terms of access time and space. Coalescing of accesses into intervals has been explored as an optimization of access history [13], [14], [15], [16], [17]. Most of these tools either use easy-to-parallelize data structures such as hashmaps and/or does not perform asynchronous access history management.

Race Detection. On-the-fly race detection for series-parallel programs is a heavily studied topic [39], [40], [1], [6], [33]. Most prior parallel on-the-fly race detection algorithms for series-parallel programs have focused on optimizing the reachability data structure and they use a hashmap-based access history [3], [33], [7]. Race detection for more general (non-series-parallel) programs has also been studied extensively [31], [29], [41], [11], [10], [9], [12]. Our work on access history optimization can plausibly be applied to more general race-detection as well. In general race detection, however, it is often not sufficient to keep two reader strands for each memory location; therefore, we have to rethink the design of the reader treaps.

Concurrent Trees. Concurrent balanced trees are widely studied both theoretically and experimentally [42], [43], [44], [45], [46] and are one plausible way of implementing our race detector. However, they are generally slow and/or difficult to implement correctly. Our design most closely resembles various software combining techniques, designed primarily to reduce concurrency overhead in concurrent data structures [47], [48], [49]. In these techniques, generally, each processor inserts a request in a shared queue and a single processor sequentially executes all outstanding requests later. We use a pull strategy instead of a push strategy in order to reduce contention on a shared queue where one of our treap workers collects all the outstanding intervals for insertion into treaps.

VI. Concluding Remarks

We have described a parallel race detector with an optimized interval-based access history. The goal of our implementation is to minimize parallelization and synchronization overhead and achieve efficient parallel performance. While we find good parallel performance compared to state-of-the-art hashmap-based access history, the scalability of some benchmarks left more to be desired, such as `mmul` and `sort` with large input sizes (i.e., Figure 4 in Section IV when running on 32 cores).

To get better performance as we increase the number of workers, we would need to parallelize the treap accesses since they are increasingly more likely to become the bottleneck. One promising strategy would be to design a treap that allows for batched insert or batched query without high synchronization overhead so that multiple strands can be processed in parallel by multiple treap workers on a single treap. In order to implement this, however, it would be important to solve the stack-memory reuse problem (discussed in Section III-F in Section III) via efficient address translation so as to prevent detection of false races. In addition, the trace data structure would have to be more complicated so as to allow us to collect multiple strands from the same worker.

References

[1] M. Feng and C. E. Leiserson, "Efficient detection of determinacy races in Cilk programs," *Theory of Computing Systems*, vol. 32, no. 3, pp. 301–326, 1999.

[2] R. H. B. Netzer and B. P. Miller, "What are race conditions?" *ACM Letters on Programming Languages and Systems*, vol. 1, no. 1, pp. 74–88, March 1992.

[3] J. Mellor-Crummey, "On-the-fly detection of data races for programs with nested fork-join parallelism," in *Proceedings of Supercomputing'91*, 1991, pp. 24–33.

[4] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav, "Efficient data race detection for async-finish parallelism," in *Runtime Verification*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, vol. 6418, pp. 368–383.

[5] J. T. Fineman, "Provably good race detection that runs in parallel," Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, MA, August 2005.

[6] M. A. Bender, J. T. Fineman, S. Gilbert, and C. E. Leiserson, "On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs," in *16th Annual ACM Symposium on Parallel Algorithms and Architectures*, 2004, pp. 133–144.

[7] R. Utterback, K. Agrawal, J. Fineman, and I.-T. A. Lee, "Provably good and practically efficient parallel race detection for fork-join programs," in *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 2016, pp. 83–94.

[8] R. Surendran and V. Sarkar, *Dynamic Determinacy Race Detection for Task Parallelism with Futures*. Cham: Springer International Publishing, 2016, pp. 368–385.

[9] Y. Xu, I.-T. A. Lee, and K. Agrawal, "Efficient parallel determinacy race detection for two-dimensional dags," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2018, pp. 368–380.

[10] K. Agrawal, J. Devietti, J. T. Fineman, I.-T. A. Lee, R. Utterback, and C. Xu, "Race detection and reachability in nearly series-parallel dags," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, January 2018.

[11] R. Utterback, K. Agrawal, J. Fineman, and I.-T. A. Lee, "Efficient race detection with futures," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '19. Washington, District of Columbia: ACM, 2019, pp. 340–354.

[12] Y. Xu, K. Singer, and I.-T. A. Lee, "Parallel determinacy race detection for futures," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, Feb. 2020, p. 217–231.

[13] C. Flanagan and S. N. Freund, "RedCard: Redundant check elimination for dynamic race detectors," in *Proceedings of the 27th European Conference on Object-Oriented Programming*. Springer Berlin Heidelberg, July 2013, pp. 255–280.

[14] Y. Peng, C. DeLozier, A. Eizenberg, W. Mansky, and J. Devietti, "SLIMFAST: Reducing metadata redundancy in sound and complete dynamic data race detection," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, pp. 835–844.

[15] J. R. Wilcox, P. Finch, C. Flanagan, and S. N. Freund, "Array shadow state compression for precise dynamic race detection," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2015, p. 155–165.

[16] D. Rhodes, C. Flanagan, and S. N. Freund, "Bigfoot: Static check placement for dynamic race detection," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2017, p. 141–156.

[17] C.-S. Park, K. Sen, P. Hargrove, and C. Iancu, "Efficient data race detection for distributed memory parallel programs," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011.

[18] Y. Xu, A. Zhou, G. Q. Yin, K. Agrawal, I.-T. A. Lee, and T. B. Schardl, "Efficient access history for race detection," in *Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX 2022)*, Jan. 2022.

[19] R. E. Tarjan, C. Levy, and S. Timmel, "Zip trees," *ACM Transactions on Algorithms*, vol. 17, no. 4, Oct. 2021.

[20] R. Seidel and C. R. Aragon, "Randomized search trees," in *ALGORITHMICA*, 1996, pp. 540–545.

[21] J. Valdes, "Parsing flowcharts and series-parallel graphs," Ph.D. dissertation, Stanford University, December 1978, sTAN-CS-78-682.

[22] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *JACM*, vol. 46, no. 5, pp. 720–748, 1999.

[23] 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*. ACM, 1998, pp. 212–223.

[24] I.-T. A. Lee, S. Boyd-Wickizer, Z. Huang, and C. E. Leiserson, "Using memory mapping to support cactus stacks in work-stealing runtime systems," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2010, pp. 411–420.

[25] T. B. Schardl, W. S. Moses, and C. E. Leiserson, "Tapir: Embedding recursive fork-join parallelism into LLVM's intermediate representation," *ACM Transactions on Parallel Computing*, vol. 6, no. 4, Dec. 2019.

[26] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007, pp. 89–100.

[27] D. Bruening and Q. Zhao, "Practical memory checking with dr. memory," in *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization*, 2011, pp. 213–223.

[28] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. USENIX Association, 2012.

[29] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic race detector for multi-threaded programs," in *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 1997.

[30] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: Data race detection in practice," in *Proceedings of the Workshop on Binary Instrumentation and Applications*. ACM, 2009, pp. 62–71.

[31] C. Flanagan and S. N. Freund, "Fasttrack: efficient and precise dynamic race detection," *SIGPLAN Not.*, vol. 44, no. 6, pp. 121–133, Jun. 2009.

[32] I. Corporation, "Intel Cilk Plus software development kit," Available at http://software.intel.com/en-us/articles/intel-cilk-plus-software-development-kit/, Dec. 2011.

[33] 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*, 2012, pp. 531–542.

[34] N. Nethercote and J. Seward, "How to shadow every byte of memory used by a program," in *Proceedings of the 3rd International Conference on Virtual Execution Environments*. ACM, 2007, pp. 65—-74.

[35] W. Cheng, Qin Zhao, Bei Yu, and S. Hiroshige, "TaintTrace: Efficient flow tracing with dynamic binary rewriting," in *11th IEEE Symposium on Computers and Communications (ISCC'06)*, 2006, pp. 749–754.

[36] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu, "LIFT: A low-overhead practical information flow tracking system for detecting security attacks," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, 2006, pp. 135–148.

[37] M. Payer, E. Kravina, and T. R. Gross, "Lightweight memory tracing," in *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX Association, Jun. 2013, pp. 115–126.

[38] Q. Zhao, D. Bruening, and S. Amarasinghe, "Umbra: Efficient and scalable memory shadowing," in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, 2010, p. 22–31.

[39] J. T. Fineman and C. E. Leiserson, "Race detectors for Cilk and Cilk++ programs," in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Springer, 2011, pp. 1706–1719.

[40] E. Pozniansky and A. Schuster, "Efficient on-the-fly data race detection in multithreaded C++ programs," pp. 179–190, 2003.

[41] ——, "MultiRace: Efficient on-the-fly data race detection in multithreaded c++ programs: Research articles," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 3, pp. 327–340, Mar. 2007.

[42] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[43] R. Bayer and M. Schkolnick, "Concurency of operations on b-trees," *Acta Informatica*, vol. 9, pp. 1–21, 1977.

[44] M. A. Bender, J. T. Fineman, S. Gilbert, and B. C. Kuszmaul, "Concurrent cache-oblivious B-trees," in *15th ACM Symposium on Parallel Algorithms and Architectures*, 2005, pp. 228–237.

[45] T. Johnson and D. Shasha, "The performance of current B-tree algorithms," *ACM Trans. Database Syst.*, vol. 18, no. 1, pp. 51–101, 1993.

[46] A. Braginsky and E. Petrank, "A lock-free B+tree," in *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*, 2012, pp. 58–67.

[47] P. Fatourou and N. D. Kallimanis, "Revisiting the combining synchronization technique," in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 2012, pp. 257–266.

[48] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir, "Flat combining and the synchronization-parallelism tradeoff," in *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, 2010, pp. 355–364.

[49] Y. Oyama, K. Taura, and A. Yonezawa, "Executing parallel programs with synchronization bottlenecks efficiently," in *Proceedings of International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA '99)*, 1999, pp. 182–204.