Efficient Access History for Race Detection

Yifan Xu* Anchengcheng Zhou* Grace Q. Yin† Kunal Agrawal*
I-Ting Angelina Lee* Tao B. Schardl†

Abstract

While there has been extensive research on race-detection algorithms for task parallel programs, most of this research has focused on optimizing a particular component — namely ***reachability analysis***, which checks whether two instructions are logically in parallel. Little attention has been paid to the other important component, namely the ***access history***, which stores all memory locations previous instructions have accessed. In theory, the access history component adds no asymptotic overhead; however, in practice, it is often the most expensive component of race detection since it is queried and (possibly) updated at each memory access. We optimize this component based on the observation that, typically, strands within parallel programs access contiguous blocks of memory. Therefore, instead of maintaining the access history at the granularity of individual memory locations, we maintain it at the granularity of these (varying size) intervals. To enable this access history, we propose (1) compiler and runtime mechanisms that allow us to efficiently collect these intervals and (2) a tree-based access history data structure that allows us to update and query it at this interval granularity. The resulting tool can race detect fork-join code with amortized constant overhead, assuming the number of intervals is small compared to the total work of the computation. Our evaluations indicate that this technique improves the performance of race detection on several benchmarks.

1 Introduction

A ***determinacy race*** [9] (or a ***general race*** [19]), occurs when two or more logically parallel instructions access the same memory location in a ***conflicting*** way, i.e., at least one of the accesses is a write. In the context of the ask-parallel programming, determinacy races are often considered bugs since they can lead to nondeterministic behaviors.

*Washington University in St. Louis, xuyifan@wustl.edu, ann.zhou@wustl.edu, kunal@wustl.edu, angelee@wustl.edu.

†Massachusetts Institute of Technology, graceyin@mit.edu, neboat@mit.edu.

Researchers have proposed several algorithms for detecting determinacy races in task-parallel code [16, 9, 10, 24, 2, 11, 35, 33, 39, 1, 36, 40]. Most of this work focuses on detecting races "on the fly" as program executes for a particular input. These algorithms provide the strong guarantee that the race detector has no false positives and if the race detector does not find a race, then the program has no races for that input. Such on-the-fly race detectors consist of two important components: (1) a ***reachability analysis*** component that determines 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) component that records (possibly a subset of) strands that have accessed a given memory location in the past. When a strand s accesses a particular memory location x, the race detector first queries the access history to find prior strands that have accessed x in a conflicting way. Next, the race detector queries the reachability component to determine if any of the strands with conflicting accesses is logically in parallel with the current strand s. If so, a race is reported. If not, the access history is updated with this new strand s so future strands can detect races.

The prior work on race detection has primarily focused on designing data structures and/or runtime mechanisms for maintaining the reachability component in a provably and practically efficient manner. In contrast, the access history has received little attention. Most prior race detectors maintain access history by using an optimized hashmap to maintain the mapping from each memory address to previous accesses, which allows for (amortized) constant time insertions and queries from the access history. In practice, however, the management of access history often incurs much higher overhead than the reachability component does.

To illustrate this fact, Figure 1 shows the overhead of each component of a ***vanilla*** sequential race detector for Cilk [3, 15], a C/C++-based task parallel platform. This race detector implements SP-Order [2], a state-of-the-art algorithm which incurs constant overhead for managing reachability for fork-join parallel computations. SP-Order executes the computation sequentially;

Copyright © 2022 by SIAM
Unauthorized reproduction of this article is prohibited

based on the parallel constructs observed during execution, it maintains a reachability data structure that can answer the queries about whether two strands are logically in parallel. The vanilla race detector uses an optimized two-level page-table-like hashmap to manage access history. In Figure 1, the baseline column is the running time of the program without race detection. The reachability column shows the execution times that account for the compiler instrumentation and data structure updates to maintain the reachability data structure. The full column shows the execution times with both reachability and access history components[1], indicating that access history is the most expensive component of race detection.

In this paper, we propose mechanisms to speed up sequential race detectors for task-parallel code, focusing on optimizing the access history component. The key observation is as follows: For many task-parallel programs, a single strand typically performs many accesses to contiguous memory locations. We shall refer to a range of contiguous memory accessed by the same strand as an ***interval***. Figure 1 shows the number of distinct (four-byte) memory words read/written and the number of intervals read/written for the tested benchmarks. The number of intervals can be several magnitudes smaller than the number of memory words. If we manage access history at the granularity of intervals instead of memory words, we can reduce both the time overhead and memory footprint of the access history.

Given this observation, we propose two advances to optimize the access history. First, instead of checking races at every memory access, we wait until end of a strand and check for races on all accesses performed by the strand at this point. This allows us to perform ***temporal*** and ***spatial*** coalescing. In temporal coalescing, we remove duplicate accesses, referred to as the ***deduplication*** — if the strand accesses the same memory location again and again, we only check for races and record this access once at the end of the strand, thereby reducing the number of queries to the access history and reachability data structures. In spatial coalescing, we coalesce contiguous memory accesses within a strand into intervals and invoke the access history and reachability data structures at the interval granularity.

Our race detector performs coalescing at both compile time and runtime. Some spatial coalescing occurs at compile time when the compiler can statically detect that the memory accesses within a strand are contiguous. Doing so allows the race detector to lower the instrumentation overhead, since instrumentation (i.e., invocations to the race detector) occurs at the granularity of intervals as opposed to at every memory access. The compile-time coalescing is conservative, however, and may miss coalescing opportunities. Our detector at runtime checks for additional opportunities for coalescing. Collectively, compile-time and runtime coalescing allow us to exploit spatial and temporal locality that exist in the code to reduce overheads due to instrumentation and calls to access history.

The second advance is in access history data structure. Instead of storing accesses at word granularity in a hashmap, we store them as intervals. Doing so allows the access history to be represented in a more compact fashion, but we need a data structure that allows for efficient updates and queries of intervals. Given an interval to insert (or query), we must find all overlapping intervals already in the data structure efficiently.

We use a balanced binary search tree data structure to maintain the access history. (Our implementation uses treaps [34, 30], but any balanced binary search tree would work.) Our construction differs from normal interval trees since it enforces that no two intervals within the tree overlap and allows one to quickly identify all overlapping intervals. In particular, the cost of inserting and querying in our data structure for an interval x is $O(h + k)$ where h is the height of the tree and k is the number of intervals that overlap with x. By maintaining a balanced binary search tree such as a treap, our insert and query cost is bounded by $O(\lg n + k)$ (with high probability), where n is the number of intervals in the treap when x is inserted. This leads us to the overall computation time as follows: Given a computation with T_1 work — the time it takes to execute the computation on one processor — our race detector runs in $O(T_1 + n \lg n)$ time, where n is the number of intervals generated by the program. If n is small compared to T_1, which is typically the case, our race detector can race detect the computation in $O(T_1)$ time, incurring amortized constant overhead.

We have developed a race detector for task-parallel code based on this design, called STINT (Sequential Treap-based INTerval race detector).[2] Experiments suggest that our optimizations are beneficial. Compared to the vanilla system, which has an average overhead (geometric means) of 78.13×, STINT incurs an average overhead (geometric means) of 18.61×, which is a 4× improvement. We also analyzed the treap operation overhead in detail, and found that a treap operation overhead tends to be dominated by the tree height as the number of overlap intervals tends to be really

[1] The access history part includes queries to the reachability component, as the tool checks for races as it updates the access history.

[2] Code can be obtained at `https://github.com/wustl-pctg/STINT`.

Copyright © 2022 by SIAM
Unauthorized reproduction of this article is prohibited

	base	vanilla reach.	vanilla full detection	# accesses $\times 10^6$ read	# accesses $\times 10^6$ write	# intervals $\times 10^6$ read	# intervals $\times 10^6$ write
chol	0.61	0.61 (1.00×)	84.66 (139.78×)	1466.0	671.2	2.1	0.7
fft	13.55	13.59 (1.00×)	488.19 (36.03×)	2013.9	1400.9	325.4	16.3
heat	4.36	4.34 (1.00×)	367.24 (84.23×)	5274.3	1053.8	2.2	1.0
mmul	8.07	8.12 (1.00×)	355.66 (44.07×)	17712.5	536.9	33.6	8.4
sort	3.39	3.41 (1.00×)	72.27 (21.32×)	693.7	535.1	1.3	0.2
stra	1.49	1.50 (1.00×)	423.43 (284.18×)	3173.5	342.0	2.1	0.8
straz	1.54	1.54 (1.00×)	244.54 (158.79×)	3814.0	216.4	4.5	1.7

Figure 1: Overheads of a vanilla race detector. Time shown in seconds. The first four columns from left to right show the benchmark name, its running time without race detection, that with only the reachability component, and that with the full race detection. The numbers in parenthesis show the overhead comparing to the baseline. The last four columns show the number of memory locations and intervals accessed, on the order of millions.

small. Moreover, since the treap overhead is small compared to other operations performed by STINT, the race detector overhead remains stable as the number of intervals increases.

2 Preliminaries

This paper focuses on fork-join parallelism, which creates a particular class of computation, called ***series-parallel DAG*** [37]. STINT utilizes ***SP-Order*** [2], a sequential race detection algorithm for fork-join parallel programs to perform reachability. This section briefly explain these concepts.

Fork-Join Parallelism. Languages and libraries that enable fork-join parallelism typically provide two main keywords for expressing logical parallelism: `spawn` and `sync`. The `spawn` keyword precedes a subroutine call and denotes that the spawned subroutine is allowed to execute in parallel with the continuation of the parent. The `sync` keyword ensures that all previously spawned subroutines must return before the control can pass the `sync` statement.[3]

The DAG Model. Parallel computations can be modeled as ***directed-acyclic graphs*** (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. For a pair of nodes in a DAG, they are ***in series*** if there is a path from one to the other, and are ***in parallel*** otherwise. Fork-join computations generate a special class of DAGs called ***series-parallel DAGs*** [37] (or ***SP DAGs*** for short) that can be constructed by repeatedly using series and parallel compositions.

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. When the execution successfully passes a `sync`, it creates a ***sync node*** with multiple incoming edges. Thus, sequential execution of the computation corresponds to a depth-first left-to-right (i.e., spawned subroutine first) traversal of the DAG, henceforth referred to as the ***sequential order***. Based on this convention, we say that a node a is to the ***left-of*** node b if either 1) a is in parallel with b and precedes b in the sequential order, or 2) a is in series with b and follows b in the sequential order.

Race Detecting SP DAGs. We utilize the SP-Order algorithm [2] which maintains the reachability by remembering executed strands in two total orders: the ***English order*** that follows the sequential order and the ***Hebrew order*** that mirrors it (i.e., depth-first and right-to-left). Given an SP DAG, maintaining two such orders suffices to perform reachability analysis. In prior work, Feng and Leiserson [9] showed that to race detect fork-join code *sequentially*, it suffices to store the left-most reader and the last writer (in sequential order) for each memory location. SP-Order enables us to check for whether a new reader is left-of the previous reader in constant time.

3 Compile-Time and Runtime Coalescing

This section describes the proposed compile-time and runtime coalescing. The compile-time coalescing can decrease instrumentation overhead, and the runtime coalescing reduces the number of intervals and provides the additional benefit of deduplication.

3.1 Compile-Time Coalescing To perform compile-time coalescing, STINT uses the Tapir compiler [29] and leverages its representation of task parallelism. Although the details of how the compiler performs coalescing are beyond the scope of this paper, we examine at a high level what coalescing the compiler can and cannot do.

Algorithm 1 presents a pseudocode example of compile-time coalescing for the base case of the matrix-multiplication code (`mmul`) from the Cilk-5 distribu-

[3] Languages differ in exact semantics and keywords, but most task parallel languages support such fork-join parallelism.

Copyright © 2022 by SIAM
Unauthorized reproduction of this article is prohibited

Algorithm 1: Base case of `matmul`

```
   Data: Submatrices A, B, and C, of size m × n, n × p,
         and m × p respectively, where each submatrix
         lies inside a larger N × N matrix that is stored
         in row-major order.
   Result: C ← C + A · B
1  for i ← 0 to m do
2      __coalesced_load_hook(C[i · N], p);
3      __coalesced_store_hook(C[i · N], p);
4      for j ← 0 to p do
5          t ← load(C[i · N + j]);
6          __coalesced_load_hook(A[i · N], n);
7          for k ← 0 to n do
8              a ← load(A[i · N + k]);
9              __load_hook(B[k · N + j]);
10             b ← load(B[k · N + j]);
11             t ← t + a ∗ b;
12         store(C[i · N + j], t);
```

tion [14]. The `mmul` benchmark performs dense matrix-matrix multiplication on matrices stored in row-major order using a parallel recursive divide-and-conquer algorithm. This algorithm divides the input matrices along the longest dimension and recursively multiplies the resulting rectangular submatrices. The base case of this recursion multiplies small rectangular submatrices serially, using the pseudocode in Algorithm 1, where the `load` and `store` functions denote hardware operations to load and store memory, respectively.

To enable race detection, the compiler inserts calls to the `__load_hook`, `__coalesced_load_hook`, and `__coalesced_store_hook` hook functions to identify memory accessed. In the `__coalesced_load_hook` and `__coalesced_store_hook` functions, the first argument identifies the starting memory address loaded or stored, and the second argument specifies the amount of memory accessed. The `__coalesced_load_hook` and `__coalesced_store_hook` functions in particular identify coalesced instrumentation that the compiler inserted. For didactic simplicity, this pseudocode assumes that a single element of the matrix has size one.

As Algorithm 1 shows, the compiler is able to insert coalesced instrumentation for accesses to the C and A submatrices. The compiler justifies representing accesses to C using coalesced loads and stores on lines 2 and 3 as follows. Each iteration of the j loop (lines 4–12) loads and stores memory location $C[i \cdot N + j]$. Hence, one invocation of the j loop loads and stores all of memory from $C[i \cdot N]$ up to, but not including, $C[i \cdot N + p]$. In addition, because this base case is serial, these loads and stores cannot race with any loads or stores within the same invocation of the base case. Hence, it is equivalent to represent accesses in the j loop to individual elements of C as coalesced accesses before the j loop to the memory from $C[i \cdot N]$ to $C[i \cdot N + p]$. In other words, a determinacy race will exist with a coalesced access to this range of memory addresses if and only if a determinacy race exists with a load or store to an individual element of C in the j loop. A similar analysis allows the compiler to represent the accesses to A with a coalesced-load on line 6.

Algorithm 1 also shows an existing limitation of the compiler's ability to coalesce instrumentation. In particular, line 9 shows that the compiler does not coalesce instrumentation for loads from the B matrix (line 10). In this code, the k loop (lines 7–11) reads the B submatrix in column-major order. But because the B matrix is stored in row-major order, the reads from B in the k loop do not cover contiguous memory locations. Hence, the compiler's analysis of the `load` operation on line 10 in the context of the k loop does not allow it to generate coalesced instrumentation for these loads. As a result, the compiler simply instruments the `load` on line 10 directly, using a call to `__load_hook` on line 9.

Algorithm 2: Insertion-sort base case of `cilksort`

```
   Data: Pointers l and h into an array A of n integers
   Result: Integers between l and h are sorted
1  q ← l + 1;
2  while q ≤ h do
3      a ← load(q); p ← q − 1;
4      while p ≥ l do
5          b ← load(p);
6          if b > a then store(p + 1, b) ;
7          else break ;
8          p ← p − 1
9      store(p + 1, a);
```

3.2 Runtime Coalescing While more sophisticated compiler analysis can reveal additional opportunities to coalesce, the inherent limitations of compile-time coalescing motivate runtime coalescing. Not only can the runtime coalesce accesses to matrix B shown in Algorithm 1 but it can also coalesce intervals that depend on the input, such as the insertion sort for the base case of the `sort` benchmark, which may repeatedly store to the same range of memory locations between two pointers, as shown in Algorithm 3. In this base case, multiple executions of the inner loop (lines 4–8) may repeatedly store to the same range of memory locations between the pointers l and h. But because the store on line 6 is predicated on the comparison of input values on line 6, the compiler cannot statically determine the range of memory locations that this base case will store to. In contrast, runtime coalescing can identify these overlapping ranges and coalesce them.

To perform runtime coalescing, we use a ***bit***

Copyright © 2022 by SIAM
Unauthorized reproduction of this article is prohibited

hashmap to keep track of which memory locations are accessed during a strand's execution. The bit hashmap is a compact version of the access history hashmap used by vanilla described in Section 1. Specifically, we use two separate two-level page-table like hashmaps to perform runtime coalescing: one for read accesses and one for write accesses. When an access is made, the prefix and suffix of its address are used to index into the first- and second-level tables, respectively. Tables at the second level are initialized lazily on first access. Each second-level table contains an array of 64-bit integers, where each bit represents a four-byte range. A bit is set if the corresponding word is accessed within the current strand and unset otherwise.

Runtime coalescing exploits the fact that the compiler performs some coalescing. When a coalesced load or store hook executes, the setting of the corresponding bits are done using bit tricks that employ bit-level parallelism. As the hashmap tends to be sparsely populated, vectors are used to remember indices corresponding to the first and second-level table entries set within the strand. After the strand finishes, we iterate through the stored indices to compute the intervals accessed and clear out the table entries for the next strand.

Runtime coalescing provides multiple benefits. First, runtime coalescing directly observes the program execution and can discover opportunities due to input-dependent or pointer-based operations that the compiler struggles to analyze. Second, overlapping intervals generated at two different points in the same strand are merged into a single interval. Finally, runtime coalescing provides ***deduplication***: multiple accesses to the same memory location within a strand are coalesced into one, which incurs a single update / query to the access history as opposed to multiple. In contrast, the vanilla race detector checks for races at each access. Even though the repeated memory accesses will generate repeated updates to the runtime coalescing bit hashmaps, updates on the bit hashmaps are significantly cheaper than those on the hashmap access history used in vanilla, because the hashmap access history keeps track of much more data in order to perform race detection. As we shall see in Section 5, both compile-time and runtime coalescing provide benefit, but the runtime coalescing provides greater benefit due to these reasons.

4 Interval-Based Access History

We now describe the access history data structure that efficiently supports (1) query to find all intervals that potentially conflict with a given interval; and (2) update to the data structure to insert the new interval. We also analyze its theoretical performance.

Recall that in a sequential race detector for fork-join parallelism, it suffices for each memory location to store its last writer and left-most reader [9]. In a traditional access history data structure, when a strand s writes to this memory location ℓ, we check if s is in parallel with the left-most reader of ℓ or with the last writer and declare a race if so. The strand s is now stored as the last writer of this location. Similarly, if s reads this memory location, we check if s is in parallel with the last writer and declare a race if so. We then check if s is left-of the existing left-most reader and store s as the left-most reader if so.

We want to store intervals instead of individual memory locations in the access history. We keep separate data structures for read intervals and write intervals. Each of these will store interval objects, say x with three fields: $x.start$ and $x.end$ denote the beginning (inclusive) and end (exclusive) of the interval and $x.accessor$ stores the strand we want to store — the last writer for the write data structure and the left-most reader for the read data structure. When convenient, we denote an interval as three-tuple: $[\texttt{start}, \texttt{end}, \texttt{accessor}]$. The intervals stored within each data structure must be disjoint from each other since each memory location can have at most one last writer and one left-most reader.

When a new strand s generates an interval, it is represented as an interval object o with the appropriate *start* and *end* values and accessor s. We must check if any access within o races with any pre-existing access in the access history (if so, report a race) and then update the access history. However, this is not straightforward. Consider the following example. Say we had the following read intervals: $[8, 16, a], [24, 32, b], [40, 52, c], [52, 60, d]$. We get a new read interval $[12, 56, e]$. The tree after the update depends on the relationship of e with all other intervals. Say e is left of a and c, but not b and d. After the update, the data structure must store $[8, 12, a], [12, 24, e], [24, 32, b], [32, 52, e], [52, 60, d]$. Thus, a new interval may overlap with many previous intervals and some previous overlapping intervals may remain entirely, and some may be removed or trimmed.

Intervals are stored in two binary search trees (one each for read and write intervals) keyed by the *start* field of the interval. The data structure is similar to interval trees [6][Chp.14.3]; however, we enforce the additional ***non-overlapping*** property that all intervals in the tree must be disjoint. The two trees behave a little differently since the accessor for each interval must be the last writer in the write tree and the left-most reader in the read tree.

Say we are processing strand x with accessor s. Since the strands are processed by the race detector

Copyright © 2022 by SIAM
Unauthorized reproduction of this article is prohibited

in sequential order, all previous intervals already in the tree are "before" s in sequential order. Therefore, if x overlaps any pre-existing interval in the write tree, then x is kept since s is always the last writer and the old interval is trimmed or removed. As we saw in the example above, this is not true in the read tree since s may not be the left-most reader for all memory locations in x. Therefore, when we see an overlap, we must check whether the old reader or the new reader is the left-most reader. We will first describe how we insert an interval in the write tree and then the read tree. We then describe how we do queries.

4.1 Updating the Write Tree Given a tree T (as a pointer to the root) and a write interval x, we will use a recursive procedure to update the tree to reflect the accesses represented by interval x. We will remove/trim all intervals that overlap with x to maintain the invariant that no intervals overlap with each other in the tree.

The procedure is illustrated in Figure 2. There are two main procedures, InsertWriteInterval and RemoveOverlap. The InsertWriteInterval is the main procedure that is called at the root of the tree. The procedure is called recursively as we walk down the tree. When we are at a particular tree node, say y in the tree and trying to insert node x, we can be in the following 4 cases.

A. **No overlap:** As shown in Figure 2(A), when y and x don't overlap, we simply recurse down to one of its children. In particular, if x is completely to the right of y ($y.end \leq x.start$), no intervals in y's left subtree can overlap with x since they all end before $y.start$. However, some intervals in y's right subtree may overlap with x, so we recurse by calling InsertWriteInterval$(y.right, x)$. If $y.right$ is empty, we simply insert this interval at this leaf.

B. **Partial overlap:** Figure 2(B) illustrates the operations when x partially overlaps with y and is to the right of y ($y.start < x.start$, $x.start < y.end < x.end$). In this case, $y.accessor$ is the last writer for part of the old interval (from $y.start$ to $x.start$), but $x.accessor$ is the last writer for memory locations after it. Therefore, we set $y.end = x.start$. Again, no intervals in the left subtree of y can intersect with x, and we recurse on the right subtree of y by calling InsertWriteInterval$(x, y.left)$. A symmetric procedure is used when x is to the left of y for both this case and the previous case.

C. **Full overlap; old interval y bigger:** Figure 2(C) illustrates the operation when y fully encompasses x ($y.start \leq x.start$ and $y.end \geq x.end$). We have up to three intervals $[y.start, x.start, y.accessor]$, $[x.start, x.end, x.accessor]$, $[x.end, y.end, y.accessor]$.[4] We keep any one of these intervals at this location in the tree (replacing the old y) and recurse down the tree to insert the other two intervals. In Figure 2 we keep the middle interval and insert the left and right intervals. Note that none of these intervals overlap with any other interval in the tree since they collectively made up y which was already in the tree before and didn't overlap with anything. Therefore, we will fall into case A from now on out — we just walk down the tree and insert in the appropriate leaf.

D. **Full overlap; new interval x bigger:** Figure 2(D) illustrates case where x fully encompasses y. Now y can be removed from the tree entirely and replaced with x. However, there may be more intervals in both the left and right subtrees of y which also overlap with x. Therefore, we use a function called RemoveOverlap to find and remove/trim these intervals. There are two versions of this function: RemoveOverlapLeft(y, x) which is called on the left subtree and RemoveOverlapRight(y, x) which is called on the right subtree. This function is illustrated separately in Figure 3 and explained below.[5]

We now describe RemoveOverlapLeft(T, x). (RemoveOverlapRight(T, x) is symmetric.) Recall that RemoveOverlapLeft(z, x) is first called when a newly inserted interval x replaced an interval y which was fully within x and z was the left child of y. The general invariant is that RemoveOverlapLeft(z, x) is called on a node z when x has been inserted into some ancestor of z to z's right (therefore, $x.end \geq z.end$) and the purpose is to find and remove/trim intervals that overlap with x. This is also a recursive function and a subset of its cases are illustrated in Figure 3. Note that there are fewer cases since z cannot fully encompass x due to the invariant of this function. We also show $parent(z)$ (the old y in the example used in InsertWriteInterval) in these figures for two reasons. First, as we will see soon, we need it for one of the cases. More importantly, we wanted to point out that even though RemoveOverlapLeft is initially called on the left child, as we make recursive calls, it can be eventually called on a right child — the function remains unchanged regardless.

A. **No overlap:** If x and z don't overlap with x to the right of z ($z.end < x.start$; x cannot be to the left of z for RemoveOverlapLeft due to the invariant

[4] There may be fewer than 3 intervals if one or both end points are equal for x and y; this is easily handled as a special case.

[5] We can (optionally) trim x so that it ends at $y.start$ when calling RemoveOverlap. The explanation is easier without, however.

Copyright © 2022 by SIAM
Unauthorized reproduction of this article is prohibited

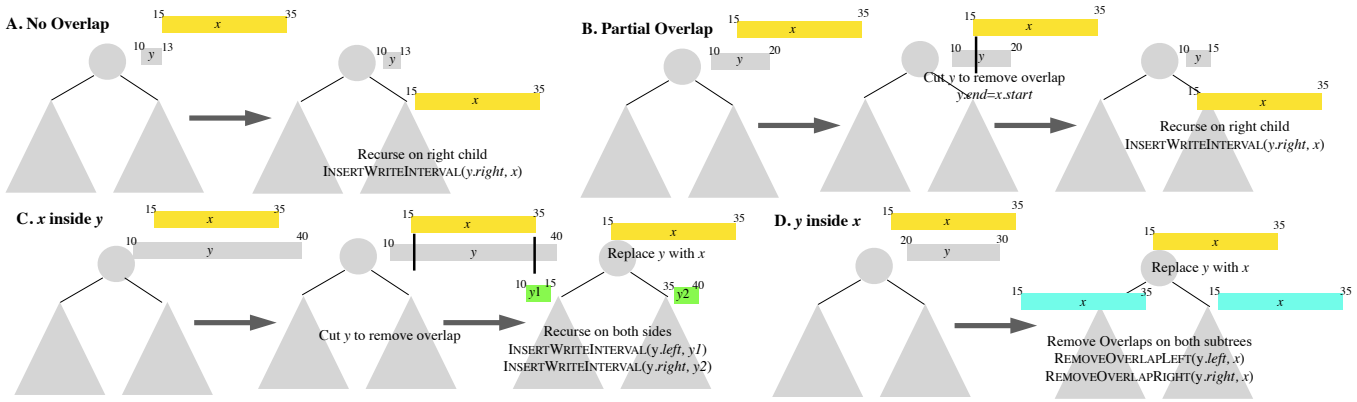

Figure 2: All cases illustrating INSERTWRITEINTERVAL(y, x) — assumes and maintains the no-overlap invariant.

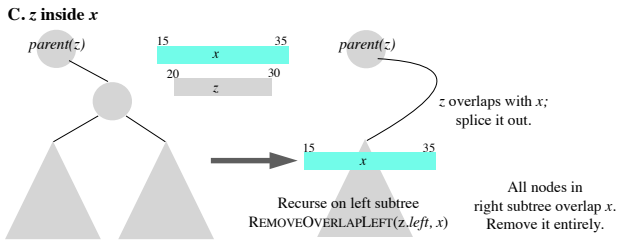

Figure 3: Case C of REMOVEOVERLAPLEFT(z, x). x was inserted at an ancestor to the right of z.

stated above), there can be no overlap in the left subtree of z. Therefore, we recurse on the right subtree by calling REMOVEOVERLAPLEFT$(z.right, x)$.

B. **Partial overlap:** If x and z partially overlap ($z.start < x.start < z.end$), we trim the interval z by setting $z.end = x.start$. Again, the left subtree of z cannot overlap with x. Since x is some ancestor of z to the right, the entire right subtree of z must now overlap with x and can be removed, thereby terminating the recursion.

C. **Full overlap** Figure 3(C) illustrates the case where x fully encompasses z. Again, the entire right subtree of z must overlap with x and is removed. In addition z itself is spliced out and replaced with its left subtree by changing the child pointer for $parent(z)$ and the parent pointer of $z.left$. In addition, we recurse on the left subtree of z by calling REMOVEOVERLAPLEFT$(z.left, x)$ to find any additional intervals that may intersect with x.

4.2 Inserting an Interval in the Read Tree

Maintaining the read tree is more complicated. In a read tree, when the new interval x overlaps with some old interval y, we may keep the y if it is left-of x. As seen in the example at the beginning of the section, this can lead to some intervals being removed and trimmed

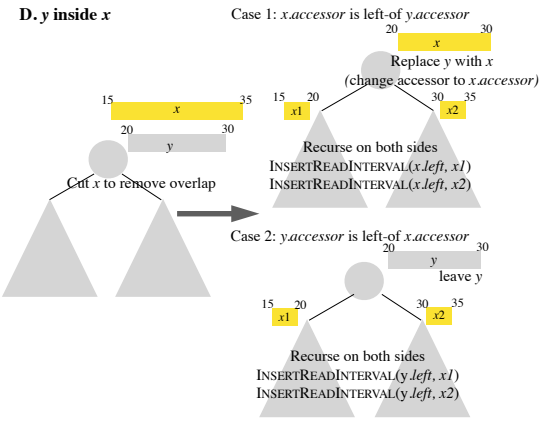

Figure 4: Case D of INSERTREADINTERVAL(y, x).

while the new interval may also be trimmed in many pieces. We have similar cases as the write tree, but the cases are handled differently. As with the write tree, we only show one direction — where x is to the right of y — the other case is symmetric.

A. **No overlap:** This case is identical to the write tree — we simply recurse to the appropriate subtree.

B. **Partial overlap:** The case of partial overlap is slightly more complicated. We have two cases. If the new accessor $x.accessor$ is left-of the old accessor $y.accessor$, then the accessor for $[y.start, x.start]$ is old $y.accessor$ but the accessor for $x.start$ onwards is the new $x.accessor$. Therefore, this case is handled like the write tree — we cut the old interval y down by setting $y.end = x.start$ and recurse to the right subtree by calling INSERTREADINTERVAL$(y.right, x)$. If, on the other hand, the old interval's accessor $y.accessor$ is left-of $x.accessor$, then we must keep the entire interval y intact. In this case, we trim x by setting $x.start = y.end$ and use this modified interval x to

Copyright © 2022 by SIAM
Unauthorized reproduction of this article is prohibited

recurse to the right subtree.

C. **Full overlap; old interval y bigger:** This is the easiest case. If the new interval is left of the old interval, then the read tree behaves like the write tree — the old interval is cut into three portions, one of the portions is kept at this location, and the other two are inserted with guaranteed no further overlap. If the old interval is left of the new interval, we just keep the old interval; nothing changes and we are done.

D. **Full overlap; new interval x bigger:** As illustrated in Figure 4, the case where x fully encompasses y is the most different from the write tree since we cannot simply remove y. First, y might be left of x and therefore must be kept. Even more importantly, there may be other intervals within y's subtrees that overlap with x and have accessors left of x. Therefore, we cut x into three pieces. The middle portion stays here and is labeled with $x.accessor$, if $x.accessor$ is left-of $y.accessor$, or $y.accessor$ otherwise. The other two portions are inserted into the left and right subtrees by recursing.

4.3 Queries to Check for Races In order to check for races with interval x, we must find intervals that overlap with x. Note that the procedure InsertWriteIntervals already finds all overlapping intervals as it walks down the tree. [6] The main wrinkle is that when we are in case B or C of RemoveOverlap and remove entire subtrees, we must walk through those subtrees to check for races with all intervals in that subtree. In summary, the race detection procedure works as follows. For a write interval x, first check for races in the read tree by using a procedure similar to InsertWriteInterval, but making no modifications to the tree itself. Then insert into the write tree while checking for races as we go. For a read interval x, first check for races in the write tree by using a procedure similar to InsertWriteInterval, but making no modifications to the tree. Then insert into the read tree by using InsertReadInterval.

4.4 Performance Analysis We have described the algorithm with a generic binary search tree. In order to keep the height low, we use a balanced binary search tree such as a treap which has height $O(\lg m)$ with high probability if there are m nodes in the treap.

We first bound the number of intervals that can be in the data structure at any given time. For the write tree, one can easily see that InsertWriteInterval increases the number of intervals by at most 2 since it doesn't create splits. For the read tree, in principle, the number of intervals in the tree can double with a single insertion. We use an amortization argument to argue that it cannot happen all the time.

Lemma 4.1. *When* InsertWriteInterval *(resp.* InsertReadInterval*) has been called on the root of the write tree (resp. read tree) m times, then the total number of intervals in the respective tree is $O(m)$.*

Proof. We first look at the easier case of the write tree. First, note that RemoveOverlap doesn't add any new intervals, only removes or trims existing ones and neither does case A for InsertWriteInterval. Cases B and D also just trim intervals, but do not add new ones. The only way a new interval is added is (a) x reaches a leaf node in case A and gets added (adding only one interval); or (b) in case C, we split an existing interval y and insert y_1 and y_2. In this case y_1 and y_2 are guaranteed to have no overlaps and get added at the leaves, causing an additional two intervals. Therefore, every time we insert a new interval, we add at most two additional intervals to the tree.

Now consider the more complicated case of the read tree. Again, just like the write tree, cases A–C do not add intervals to the tree. However, case D is interesting, because we call InsertReadInterval on both subtrees. There is no guarantee that these new x_1 and x_2 won't also overlap with additional intervals further down the tree and subdivide further. In the worst case, if we had i intervals before a particular interval was added, we can have $2i+1$ intervals after it was added. Consider the following example: say we had $[1, 2, a]$, $[3, 4, b]$, and $[5, 6, c]$ in the tree. If we read an interval $[0, 7, d]$ where a, b, c are all left-of d, our tree will contain $[0, 1, d]$, $[1, 2, a]$, $[2.3.d]$, $[3, 4, b]$, $[4, 5, d]$, $[5, 6, c]$, and $[6, 7, d]$.

However, it turns out that the total number of intervals cannot double with every insertion. We will see this by counting not just intervals, but also ***gaps***. Gaps are memory ranges between consecutive intervals — in our example before d is inserted, $[0, 1]$, $[2, 3]$, $[4, 5]$, $[5, -]$ are gaps. When we insert an interval that doesn't overlap with any existing interval, we increase the number of intervals by exactly one and we increase the number of gaps by at most one, for a collective increase of at most 2 (we may not increase the number of gaps if the new interval is right next to another or decrease the number of gaps by one if we fill in the gap between two intervals). When we insert an interval that overlaps other intervals, we may increase the number of intervals by a lot, but only by filling in gaps. Therefore, the collective increase in the number of gaps and intervals is at most two in all cases. An empty tree has one gap. Since each insert increases the

[6] In fact, InsertWriteInterval also finds all overlaps — InsertWriteInterval is more efficient, however.

Copyright © 2022 by SIAM
Unauthorized reproduction of this article is prohibited

number of gaps and intervals (collectively) by at most 2, the total number of intervals is at most $2m+1$. □

Now we can bound the total insert and query time. Again, it is relatively straightforward to see that queries take $O(h+k)$ time for write intervals since the procedure involves walking down the tree and checking every overlapping interval. For the read tree, we must again do an amortization argument based on the overlapping intervals.

LEMMA 4.2. *Inserting an interval and querying into the access history takes $O(h+k)$ time where h is the height of the larger tree (read or write) and k is the number of intervals that overlap with x across both trees.*

Proof. We first look at the easier case of the write tree. First, note that REMOVEOVERLAP doesn't add any new intervals, only removes or trims existing ones and neither does case A for INSERTWRITEINTERVAL. Cases B and D also just trim intervals, but do not add new ones. The only way a new interval is added is (a) x reaches a leaf node in case A and gets added (adding only one interval); or (b) in case C, we split an existing interval y and insert y_1 and y_2. In this case y_1 and y_2 are guaranteed to have no overlaps and get added at the leaves, causing an additional two intervals. Therefore, every time we insert a new interval, we add at most two additional intervals to the tree.

Now consider the more complicated case of the read tree. Again, just like the write tree, cases A–C do not add intervals to the tree. However, case D is interesting, because we call INSERTREADINTERVAL on both subtrees. There is no guarantee that these new x_1 and x_2 won't also overlap with additional intervals further down the tree and subdivide further. In the worst case, if we had i intervals before a particular interval was added, we can have $2i+1$ intervals after it was added. Consider the following example: say we had $[1,2,a]$, $[3,4,b]$, and $[5,6,c]$ in the tree. If we read an interval $[0,7,d]$ where a,b,c are all left-of d, our tree will contain $[0,1,d]$, $[1,2,a]$, $[2.3.d]$, $[3,4,b]$, $[4,5,d]$, $[5,6,c]$, and $[6,7,d]$.

However, it turns out that the total number of intervals cannot double with every insertion. We will see this by counting not just intervals, but also ***gaps***. Gaps are memory ranges between consecutive intervals — in our example before d is inserted, $[0,1]$, $[2,3]$, $[4,5]$, $[5,-]$ are gaps. When we insert an interval that doesn't overlap with any existing interval, we increase the number of intervals by exactly one and we increase the number of gaps by at most one, for a collective increase of at most 2 (we may not increase the number of gaps if the new interval is right next to another or decrease the number of gaps by one if we fill in the gap between two intervals). When we insert an interval that overlaps other intervals, we may increase the number of intervals by a lot, but only by filling in gaps. Therefore, the collective increase in the number of gaps and intervals is at most two in all cases. An empty tree has one gap. Since each insert increases the number of gaps and intervals (collectively) by at most 2, the total number of intervals is at most $2m+1$. □

THEOREM 4.1. *Across the entire computation, the total cost of checking for races is $O(n \lg n + T_1)$ where n is the total number of intervals generated by the program and T_1 is the work.*

Proof. The total number of intervals in either tree never exceeds $O(n)$ from Lemma 4.1. Therefore, from Lemma 4.2, the cost of each individual interval is $O(\lg n + k)$ if we use a balanced tree. If an interval overlaps k other intervals, then it must have size at least k and therefore, the program must do k work to generate this interval. Therefore, over all intervals, the total cost of race detection is $O(n \lg n + T_1)$ where $n \lg n$ term comes from adding the $\lg n$ cost over n intervals and T_1 comes from adding k over all intervals. □

5 Empirical Evaluation

In this section, we empirically evaluate STINT, the race detector that incorporates the optimizations described in Sections 3 and 4 and their impact on performance. Experiments suggest that our optimizations are beneficial. Compared to the vanilla system, which has an average overhead (geometric means) of 78.13×, STINT incurs an average overhead (geometric mean) of 18.61×. Detailed analysis indicates that the overhead of a single treap operation is dominated by the tree traversal as the number of overlapping intervals tend to be small. Moreover, since the treap overhead is small compared to other operations performed by STINT, STINT overhead stays constant as the number of intervals increases.

Experimental Setup. We used seven standard task-parallel benchmarks (where b indicates base-case size and other parameters describe the input size): Cholesky decomposition (`chol`, $n = 2000, z = 20000, b = 16$); parallel mergesort (`sort`, $n = 2.5e^7, b = 2048$); 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 ($n = 2048, b = 64$).

All experiments were run on a machine with two 20-core Intel Xeon Gold 6148 processors, clocked at

Copyright © 2022 by SIAM
Unauthorized reproduction of this article is prohibited

2.40 GHz, with hyperthreading disabled. Each core has separate private 32 KB L1 data and 32 KB L1 instruction caches, and a 1 MB private L2 cache. Each socket has a 27.5 MB shared L3 cache. The machine has 768 GB of main memory. All software is compiled with the OpenCilk [28] compiler, based on Tapir [29], with `-O3` optimizations and run on Linux kernel version 4.15. We modified the compiler to perform compile-time coalescing as discussed in Section 3. Each data point is the average of 5 runs with standard deviation $< 4\%$ unless stated otherwise.

Overview of Results. We ran the benchmarks with the following four versions of the race detector to tease out the impact of each optimization.

- ***vanilla*** employs an optimized two-level page-table like hashmap to manage access history and uses a compiler that generates instrumentation for each memory access.
- ***compiler*** introduces the compile-time coalescing discussed in Section 3.1, with the same hashmap to manage access history as in vanilla.
- ***comp+rts*** includes both compile-time and runtime coalescing discussed in Section 3 but still uses the same hashmap to manage access history.
- ***STINT*** includes both compile-time and runtime coalescing and uses the treap construction in Section 4 to manage access history.

All race detectors utilize the same implementation based on the SP-Order algorithm [2] to maintain reachability.

These different race detectors allow us to gauge the impact of each optimization. By comparing vanilla and compiler, we gauge how much instrumentation overhead is reduced. By comparing compiler and comp+rts, we gauge how much overhead the full coalescing reduces. By comparing comp+rts and STINT, we measure the impact of using a treap instead of a hashmap, which incurs higher overhead per operation but reaps the full benefit of coalescing.

	vanilla	*compiler*	*comp+rts*	*STINT*
`chol`	84.66 (138.79×)	82.87 (135.85×)	26.73 (43.82×)	19.22 (31.73×)
`fft`	488.19 (36.03×)	368.76 (27.21×)	304.92 (22.50×)	489.71 (36.14×)
`heat`	367.24 (84.23×)	326.03 (74.78×)	144.43 (33.13×)	23.24 (5.32×)
`mmul`	355.66 (44.07×)	345.08 (42.76×)	219.25 (27.16×)	220.82 (27.36×)
`sort`	72.27 (21.32×)	69.39 (20.47×)	40.63 (11.98×)	15.81 (4.66×)
`stra`	423.43 (284.18×)	414.52 (278.20×)	96.30 (64.63×)	38.33 (25.74×)
`straz`	244.54 (158.79×)	244.36 (158.68×)	100.15 (65.03×)	51.66 (33.62×)

Figure 5: Execution times (in seconds) and overheads of different versions of the race detector compared to the baseline (i.e., no race detection), whose values are shown in Figure 1.

Figure 5 shows the race-detection overhead compared to the ***baseline*** execution time, i.e., no race detection, running each version of the detector. For most benchmarks, each additional optimization brings some benefit to the overhead reduction, leading to the final result, where STINT incurs an average (geometric mean) of 18.61× overhead, much less than that of vanilla, 78.13×. The only exception is `fft`, whose overhead increased from comp+rts to STINT; we explain the reason later in the section.

Compile-Time vs. Runtime Coalescing. Now we analyze in more detail the benefit of compile-time versus runtime coalescing. The overhead decreased between vanilla and compiler but not as substantially as between vanilla and comp+rts for the following reasons. First, comp+rts is able to coalesce more. Although the access history in both comp+rts and compiler handles a given interval at four-byte granularity, the number of intervals generated is correlated with the number of top-level calls into the access history. Thus, comp+rts incurs less function-call overhead to query and update the access history. Second, comp+rts takes advantage of the runtime deduplication, which results in fewer updates to the hashmap.

To get a better sense of how much the compile-time versus runtime coalescing can do, we separately collected various memory access pattern generated by running vanilla, compiler (compile-time coalescing) and by comp+rts (both compile-time and runtime), shown in Figure 9. First, we shall examine the numbers shown on the left side of the table: the numbers of accesses / intervals generated by all three version. In some benchmarks, such as `mmul` and `heat`, the compiler was able to coalesce in a non-negligible way, but in most benchmarks, the compiler cannot coalesce as much. The runtime coalescing other the other hand, seems much more effective in coalescing and deduplicating, leading to two or three order of magnitude of decrease in the number of intervals. Moreover, the average sizes of intervals (*avg.*) tend to be a lot larger with runtime coalescing.

Another question is, how much impact does the runtime deduplication have in reducing the overhead. We can gauge the answer to this question by looking at the total bytes that made into the access history (*sum*), also shown in the table. If the runtime performs coalescing only but not deduplication, the total bytes accessed should not change from compiler to comp+rts. Thus, by comparing the total bytes accessed generated by compiler versus comp+rts, we can tell that most benchmarks benefit from deduplication.

This data, combining with the data in Figure 5, indicate that while both compile-time and runtime coalescing can be beneficial, the benefit from runtime is more significant.

Copyright © 2022 by SIAM
Unauthorized reproduction of this article is prohibited

	vanilla		*compiler*		*both*		*compiler*		*both*		*compiler*		*both*	
	acc. (r)	*acc. (w)*	*int. (r)*	*int. (w)*	*int. (r)*	*int. (w)*	*avg. (r)*	*avg. (w)*	*avg. (r)*	*avg. (w)*	*sum (r)*	*sum (w)*	*sum (r)*	*sum (w)*
`chol`	1466.0	671.2	1430.3	100.6	2.1	0.7	8.44	105.3	977.2	873.6	11510.2	10103.6	1914.5	641.9
`fft`	2013.9	1400.9	2013.8	1007.6	325.4	16.3	4.9	7.5	29.28	462.45	9474.3	7168.0	9084.6	7168.0
`heat`	5274.3	1053.8	43.2	1053.8	2.2	1.0	1946.4	15.95	9635.3	16375.9	80137.7	16032.0	20004.9	16032.0
`mmul`	17712.5	536.9	17196.5	16.8	33.6	8.4	4.1	128.0	128.0	128.0	67568.0	2048.0	4096.0	1024.0
`sort`	692.7	535.1	297.8	535.1	1.3	0.2	18.6	8.0	2256.7	13042.0	5286.1	4083.4	2870.4	2861.0
`stra`	3173.5	342.0	2665.7	342.0	2.1	0.8	16.7	12.2	1886.8	2926.6	43244.5	3967.1	3824.4	2312.7
`straz`	3814.0	216.4	3814.0	216.4	4.5	1.7	14.5	16.0	2048.0	2048.0	52708.5	3302.0	8804.5	3302.0

Figure 6: Execution statistics on memory accesses generated when running *vanilla*, with comiler coalescing (*compiler*), and with both compiler and runtime coalescing (*both*). The *acc.* and *int.* indicate the number of accesses / intervals that made into the access history, shown in millions. The *avg.* shows the average size per interval accessed and the *sum* shows the total size (in Mbytes) accessed. The *(r) / (w)* indicate read or write.

	hashmap	*treap*
`chol`	8.93	1.41
`fft`	207.72	392.50
`heat`	123.63	2.43
`mmul`	15.94	17.51
`sort`	26.36	1.54
`stra`	59.60	1.62
`straz`	52.00	3.50

Figure 7: The total time (in seconds) each benchmark spent updating its access history using hashmap (measured with comp+rts) versus treap.

Hashmap vs. Treap. Now we analyze the overhead of the treap construction in more detail and also explain why `fft` sees an overhead increase from comp+rts to STINT. We measured the time that comp+rts and STINT spent updating their respective access histories, shown in Figure 10. Indeed, the treap overhead is much larger than that of the hashmap in `fft`.

There are multiple factors at play here. Given an interval of size x, the hashmap needs to perform $2x$ operations (insert and query). On the other hand, while the STINT can reap the benefit of coalescing fully, an update to the treap incurs $O(h+k)$ time, where h is the height of the treap and k is the number of overlaps.

It turns out that while coalescing reduces the number of intervals, the reduction for `fft`, compared to other benchmarks, is less significant, and the resulting interval size is smaller as well (data shown in Figure 9). Thus, the trade-offs made by using a treap do not work well for benchmarks with characteristics like `fft` (i.e., less reduction in the number of intervals and smaller interval size).

Analysis of Treap Overhead. To better understand the treap operation overhead, we collected more data using three representative benchmarks, `fft`, `mmul`, and `sort`, where the STINT performs worse, comparable, and better than the comp+rts version (that utilizes a hashmap), respectively. Figure 8 shows the execution times and other stats of these benchmarks running baseline (no race detection), comp+rts, and STINT on different input sizes. The execution times shown here are the average of three runs.

As the input size increases, the number of intervals n should increase as well, and we would like to understand how the overhead in STINT may grow. Given a treap operation, its overhead is $O(h+k)$ where h is the height and k is the number of overlapping intervals. In the worst case, k can be large. The data in the two rightmost columns, however, shows that k is typically small, and the overhead per treap operation is dominated by the nodes visited (bounded by $O(h)$).

Given that the treap operation is dominated by the tree height, one would expect the operation overhead grows with $O(\lg m)$ with high probability, where m is the number of nodes in the tree. As such, in the worst case, the execution time of a benchmark can increase and grow with $O(n \lg n)$, where n is the number of intervals generated during execution. Fortunately, as the numbers on the left indicate, the STINT overhead compared to the baseline (*base*) remains fairly stable across different input sizes. This may seem counterintuitive, but the numbers shown in *treap oh* offer a clue: the overhead incurred by the treap data structures is relatively small compared to the rest of the race-detector overhead such that even if the treap overhead grows, the race-detector overhead is still dominated by other operations. Consequently, the treap overhead is too small to have much impact on the final execution time. The only exception to this is `fft`, which does not work well with using treap as its access history due its execution characteristics as explained earlier.

6 Related Work

Many tools employ some form of shadow memory to store shadow values of memory locations in the program-under-test; examples include memory checkers [18, 4, 31] and race detectors [27, 32, 12, 10, 2, 7, 25, 35]. Researchers have explored ways to optimize shadow memory data structures,. Some schemes encode the shadow values to reduce the space used for each memory location [17, 27, 32, 31, 5, 23], but they tend to be tool specific and can affect the precision

Copyright © 2022 by SIAM
Unauthorized reproduction of this article is prohibited

	input	*base*	*comp+rts*	*STINT*	*hash oh*	*treap oh*	*hash ops*	*treap ops*	*# nodes*	*# overlaps*
`fft`	2^{24}	2.33	58.65 (25.17×)	80.21 (34.42×)	41.45	63.01	$2.60e^8$	$1.42e^8$	29.29	0.97
	2^{25}	5.36	125.66 (23.44×)	180.03 (33.59×)	88.44	142.81	$5.21e^8$	$2.85e^8$	28.54	0.97
	2^{26}	13.55	304.92 (22.50×)	489.71 (36.14×)	207.72	392.50	$1.22e^9$	$6.83e^8$	29.56	0.98
`mmul`	1024	1.01	27.20 (26.93×)	27.03 (26.76×)	1.82	1.65	$4.19e^7$	$1.05e^7$	16.50	0.69
	2048	8.07	219.25 (27.16×)	220.82 (27.36×)	15.94	17.51	$3.36e^8$	$8.39e^7$	19.31	0.69
	4096	65.98	1763.03 (26.72×)	1793.49 (27.18×)	122.05	152.51	$2.68e^9$	$6.71e^8$	21.54	0.70
`sort`	$5.0e^7$	7.17	88.68 (12.37×)	34.32 (4.79×)	57.99	3.63	$8.53e^8$	$7.02e^6$	36.53	1.88
	$1.0e^8$	14.99	179.12 (11.95×)	70.80 (4.72×)	115.72	7.40	$1.71e^9$	$1.45e^7$	38.67	1.90
	$2.0e^8$	31.57	389.38 (12.33×)	152.76 (4.84×)	254.27	17.65	$3.81e^9$	$3.21e^7$	40.42	1.90

Figure 8: Execution times of `fft`, `mmul`, and `sort` running on baseline (*base*, i.e., no race detection), comp+rts, and STINT on different input sizes, with overhead compared to *base* shown in parenthesis. On the right of the execution times, we also show various stats for comp+rts (using a hashmap) and STINT, where the *oh* indicates time spent on access history only, the *ops* indicates the number of hashmap / treap operations, the *# nodes* shows the average number of nodes visited per treap operation, and the *# overlaps* shows the average number of overlaps encountered per treap operation.

of the analysis. There are various versions of direct-mapped tables [5, 23, 21, 31] or multi-level translation schemes [41, 17] which provide various trade-offs in terms of access time and space. Researchers have also explored optimizations, such as vectorization, for accessing and updating the ranges of entries in the shadow memory table [17].

Coalescing of accesses into intervals has been explored as an optimization in the context of data race detectors for pthreaded code RedCard [13] and SlimFast [22] use compile-time coalescing only and use a hashtable with one location per interval. Therefore, they can coalesce together a set of variables only when the compiler can statically prove that they are accessed together in all strands (synchronization-free regions or SFRs in their terminology) so that a single key can be used to represent the set in the hashtable. In contrast, we employ runtime coalescing and our tool dynamically splits intervals as needed during runtime.

SlimState [38] and BigFoot [26] employ dynamic coalescing for (only) arrays. These tools convert an array into an object with multiple partitions with predefined access patterns (either contiguous or strided) and these partitions and their accessors are stored in shadow memory. These papers do not detail what data structures are used to store such partitions and how much overhead such an operation incurs, as execution time bound is not their primary focus. Our work focuses on coalescing memory accesses that span contiguous memory locations (i.e., intervals); it is not limited to arrays with particular access patterns. Moreover, we show that our treap construction allows for provably efficient insertion or updating of a new interval.

Work by Park et al. [20] is perhaps the most closely related to our work. Their data-race detector employs a skiplist to manage shadow memory. However, a key difference is that their detector does not remove redundant intervals. If a new interval x overlaps existing intervals y and z, after insertion of x, all three intervals co-exist in the skiplist. Our work replaces the existing intervals upon insertion of x (though x is checked against y and z). Doing so allows the insertion, update, or query of a given interval x to be done in time $O(\lg n + k)$, where k is the number of intervals in the data structure that overlap with x (in our data structure, none of these k intervals overlap with each other). Park et al.'s bound for insertion, update, and query is $O(\lg^2 n + k')$ time, where k', the number of intervals in their data structure that overlap with x; however, this k' may be much larger than the k in our bound since many of these k' intervals may overlap or even be duplicates of each other.

7 Conclusion and Future Work

We have presented an optimization of access history to speed up race detectors for task parallel programs. While we focus on series-parallel programs in this paper, our design would work out of the box in other instances, such as race detector for pipelines or 2D grids [39, 8] since it is still sufficient to store one reader and one writer for each memory location. There are several directions of future work: First, for programming constructs such as futures, it is not sufficient to store one reader per memory location [36, 1] and generalizing our shadow memory to such programs would be interesting. Second, designing a parallel race detector with our optimized access history raises some interesting challenges — primarily, how to handle concurrent accesses to the binary search tree data structure while still maintaining efficiency, both theoretically and in practice. Finally, we would like to investigate the use of our shadow memory in tools other than race detection.

Copyright © 2022 by SIAM
Unauthorized reproduction of this article is prohibited

References

[1] Kunal Agrawal, Joseph Devietti, Jeremy T. Fineman, I-Ting Angelina Lee, Robert Utterback, and Changming Xu. Race detection and reachability in nearly series-parallel dags. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, New Orleans, Louisiana, January 2018.

[2] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles 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*, pages 133–144, 2004.

[3] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, Santa Barbara, California, July 1995.

[4] Derek Bruening and Qin Zhao. Practical memory checking with dr. memory. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization*, pages 213–223, Los Alamitos, CA, USA, 2011.

[5] 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)*, pages 749–754, 2006.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[7] Intel Corporation. Intel Cilk Plus software development kit. Available at `http://software.intel.com/en-us/articles/intel-cilk-plus-software-development-kit/`, December 2011.

[8] Dimitar Dimitrov, Martin Vechev, and Vivek Sarkar. Race detection in two dimensions. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 101–110, Portland, Oregon, USA, 2015. ACM.

[9] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–11, June 1997.

[10] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. *Theory of Computing Systems*, 32(3):301–326, 1999.

[11] Jeremy 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.

[12] Cormac Flanagan and Stephen N. Freund. Fasttrack: efficient and precise dynamic race detection. *SIGPLAN Not.*, 44(6):121–133, June 2009.

[13] Cormac Flanagan and Stephen N. Freund. RedCard: Redundant check elimination for dynamic race detectors. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, pages 255–280, Montpellier, France, July 2013. Springer Berlin Heidelberg.

[14] Matteo Frigo, Charles E. Leiserson, and Keith 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. ACM, 1998.

[15] Intel® Cilk™ Plus. `https://www.cilkplus.org`, 2013.

[16] John Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of Supercomputing'91*, pages 24–33, 1991.

[17] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 65—-74, New York, NY, USA, 2007. Association for Computing Machinery.

[18] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 89–100, New York, NY, USA, 2007.

[19] Robert H. B. Netzer and Barton P. Miller. What are race conditions? *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992.

[20] Chang-Seo Park, Koushik Sen, Paul Hargrove, and Costin Iancu. Efficient data race detection for distributed memory parallel programs. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, New York, NY, USA, 2011. Association for Computing Machinery.

[21] Mathias Payer, Enrico Kravina, and Thomas R. Gross. Lightweight memory tracing. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 115–126, San Jose, CA, June 2013. USENIX Association.

[22] Yuanfeng Peng, Christian DeLozier, Ariel Eizenberg, William Mansky, and Joseph Devietti. SLIMFAST: Reducing metadata redundancy in sound and complete dynamic data race detection. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 835–844, 2018.

[23] 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)*, pages 135–148, 2006.

[24] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Efficient data race detection for async-finish parallelism. In Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Rosu, Oleg Sokolsky,

Copyright © 2022 by SIAM
Unauthorized reproduction of this article is prohibited

and Nikolai Tillmann, editors, *Runtime Verification*, volume 6418 of *Lecture Notes in Computer Science*, pages 368–383. Springer Berlin / Heidelberg, 2010.

[25] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Scalable and precise dynamic datarace detection for structured parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 531–542, 2012.

[26] Dustin Rhodes, Cormac Flanagan, and Stephen N. Freund. Bigfoot: Static check placement for dynamic race detection. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 141–156, Barcelona, Spain, 2017. Association for Computing Machinery.

[27] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic race detector for multi-threaded programs. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP)*, October 1997.

[28] Tao B. Schardl, I-Ting Angelina Lee, and Charles E. Leiserson. Brief announcement: Open Cilk. In *Proceedings of the 30th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, page 351–353. Association for Computing Machinery, 2018.

[29] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. Tapir: Embedding recursive fork-join parallelism into llvm's intermediate representation. *ACM Trans. Parallel Comput.*, 6(4), December 2019.

[30] Raimund Seidel and Cecilia R. Aragon. Randomized search trees. In *ALGORITHMICA*, pages 540–545, 1996.

[31] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX ATC*, 2012.

[32] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, New York, New York, 2009. ACM.

[33] Rishi Surendran and Vivek Sarkar. *Dynamic Determinacy Race Detection for Task Parallelism with Futures*, pages 368–385. Springer International Publishing, Cham, 2016.

[34] Robert E. Tarjan, Caleb C. Levy, and Stephen Timmel. Zip trees, 2018.

[35] Robert Utterback, Kunal Agrawal, Jeremy Fineman, and I-Ting Angelina 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*, SPAA '16, pages 83–94, Asilomar State Beach, CA, USA, 2016. ACM.

[36] Robert Utterback, Kunal Agrawal, Jeremy Fineman, and I-Ting Angelina Lee. Efficient race detection with futures. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 340–354, Washington, District of Columbia, 2019. ACM.

[37] Jacobo Valdes. *Parsing Flowcharts and Series-Parallel Graphs*. PhD thesis, Stanford University, December 1978. STAN-CS-78-682.

[38] James R. Wilcox, Parker Finch, Cormac Flanagan, and Stephen N. Freund. Array shadow state compression for precise dynamic race detection. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, page 155–165. IEEE Press, 2015.

[39] Yifan Xu, I-Ting Angelina Lee, and Kunal 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*, PPoPP '18, pages 368–380, Vienna, Austria, 2018. ACM.

[40] Yifan Xu, Kyle Singer, and I-Ting Angelina Lee. Parallel determinacy race detection for futures. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, page 217–231. ACM, February 2020.

[41] Qin Zhao, Derek Bruening, and Saman Amarasinghe. Umbra: Efficient and scalable memory shadowing. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, page 22–31, New York, NY, USA, 2010. Association for Computing Machinery.

Copyright © 2022 by SIAM
Unauthorized reproduction of this article is prohibited

A Omitted Example in Section 3

A.1 Runtime Coalescing While more sophisticated compiler analysis can reveal additional opportunities to coalesce instrumentation, the inherent limitations of compile-time coalescing motivate runtime coalescing. Not only can the runtime can coalesce accesses to matrix B shown in Algorithm 1 but it can also coalesce intervals that depend on the input. For example, Algorithm 3 presents pseudocode with input-dependent memory-access patterns that are difficult to identify at compile time. This pseudocode implements an insertion sort for the base case of the `cilksort` benchmark. In this base case, multiple executions of the inner loop (lines 4–8) may repeatedly store to the same range of memory locations between the pointers l and h. But because the store on line 6 is predicated on the comparison of input values on line 6, the compiler cannot statically determine the range of memory locations that this base case will store to. In contrast, runtime coalescing can identify these overlapping ranges and coalesce them.

Algorithm 3: Insertion-sort base case of `cilksort`

Data: Pointers l and h into an array A of n integers
Result: Integers between l and h are sorted

```
1 q ← l + 1;
2 while q ≤ h do
3 |   a ← load(q); p ← q − 1;
4 |   while p ≥ l do
5 |   |   b ← load(p);
6 |   |   if b > a then store(p + 1, b) ;
7 |   |   else break ;
8 |   |   p ← p − 1
9 |   store(p + 1, a);
```

B Omitted Proofs in Section 4

Proof of Lemma 4.1.

Proof. We first look at the easier case of the write tree. First, note that RemoveOverlap doesn't add any new intervals, only removes or trims existing ones and neither does case A for InsertWriteInterval. Cases B and D also just trim intervals, but do not add new ones. The only way a new interval is added is (a) x reaches a leaf node in case A and gets added (adding only one interval); or (b) in case C, we split an existing interval y and insert y_1 and y_2. In this case y_1 and y_2 are guaranteed to have no overlaps and get added at the leaves, causing an additional two intervals. Therefore, every time we insert a new interval, we add at most two additional intervals to the tree.

Now consider the more complicated case of the read tree. Again, just like the write tree, cases A–C do not add intervals to the tree. However, case D is interesting, because we call InsertReadInterval on both subtrees. There is no guarantee that these new x_1 and x_2 won't also overlap with additional intervals further down the tree and subdivide further. In the worst case, if we had i intervals before a particular interval was added, we can have $2i + 1$ intervals after it was added. Consider the following example: say we had $[1, 2, a]$, $[3, 4, b]$, and $[5, 6, c]$ in the tree. If we read an interval $[0, 7, d]$ where a, b, c are all left-of d, our tree will contain $[0, 1, d]$, $[1, 2, a]$, $[2.3.d]$, $[3, 4, b]$, $[4, 5, d]$, $[5, 6, c]$, and $[6, 7, d]$.

However, it turns out that the total number of intervals cannot double with every insertion. We will see this by counting not just intervals, but also ***gaps***. Gaps are memory ranges between consecutive intervals — in our example before d is inserted, $[0, 1]$, $[2, 3]$, $[4, 5]$, $[5, -]$ are gaps. When we insert an interval that doesn't overlap with any existing interval, we increase the number of intervals by exactly one and we increase the number of gaps by at most one, for a collective increase of at most 2 (we may not increase the number of gaps if the new interval is right next to another or decrease the number of gaps by one if we fill in the gap between two intervals). When we insert an interval that overlaps other intervals, we may increase the number of intervals by a lot, but only by filling in gaps. Therefore, the collective increase in the number of gaps and intervals is at most two in all cases. An empty tree has one gap. Since each insert increases the number of gaps and intervals (collectively) by at most 2, the total number of intervals is at most $2m + 1$. ∎

Proof of Lemma 4.2.

Copyright © 2022 by SIAM
Unauthorized reproduction of this article is prohibited

	vanilla		*compiler*		*both*		*compiler*		*both*		*compiler*		*both*	
	acc. (r)	*acc. (w)*	*int. (r)*	*int. (w)*	*int. (r)*	*int. (w)*	*avg. (r)*	*avg. (w)*	*avg. (r)*	*avg. (w)*	*sum (r)*	*sum (w)*	*sum (r)*	*sum (w)*
`chol`	1466.0	671.2	1430.3	100.6	2.1	0.7	8.44	105.3	977.2	873.6	11510.2	10103.6	1914.5	641.9
`fft`	2013.9	1400.9	2013.8	1007.6	325.4	16.3	4.9	7.5	29.28	462.45	9474.3	7168.0	9084.6	7168.0
`heat`	5274.3	1053.8	43.2	1053.8	2.2	1.0	1946.4	15.95	9635.3	16375.9	80137.7	16032.0	20004.9	16032.0
`mmul`	17712.5	536.9	17196.5	16.8	33.6	8.4	4.1	128.0	128.0	128.0	67568.0	2048.0	4096.0	1024.0
`sort`	692.7	535.1	297.8	535.1	1.3	0.2	18.6	8.0	2256.7	13042.0	5286.1	4083.4	2870.4	2861.0
`stra`	3173.5	342.0	2665.7	342.0	2.1	0.8	16.7	12.2	1886.8	2926.6	43244.5	3967.1	3824.4	2312.7
`straz`	3814.0	216.4	3814.0	216.4	4.5	1.7	14.5	16.0	2048.0	2048.0	52708.5	3302.0	8804.5	3302.0

Figure 9: Execution statistics on memory accesses generated when running *vanilla*, with comiler coalescing (*compiler*), and with both compiler and runtime coalescing (*both*). The *acc.* and *int.* indicate the number of accesses / intervals that made into the access history, shown in millions. The *avg.* shows the average size per interval accessed and the *sum* shows the total size (in Mbytes) accessed. The *(r) / (w)* indicate read or write.

Proof. We first look at the easier case of the write tree. First, note that RemoveOverlap doesn't add any new intervals, only removes or trims existing ones and neither does case A for InsertWriteInterval. Cases B and D also just trim intervals, but do not add new ones. The only way a new interval is added is (a) x reaches a leaf node in case A and gets added (adding only one interval); or (b) in case C, we split an existing interval y and insert y_1 and y_2. In this case y_1 and y_2 are guaranteed to have no overlaps and get added at the leaves, causing an additional two intervals. Therefore, every time we insert a new interval, we add at most two additional intervals to the tree.

Now consider the more complicated case of the read tree. Again, just like the write tree, cases A–C do not add intervals to the tree. However, case D is interesting, because we call InsertReadInterval on both subtrees. There is no guarantee that these new x_1 and x_2 won't also overlap with additional intervals further down the tree and subdivide further. In the worst case, if we had i intervals before a particular interval was added, we can have $2i + 1$ intervals after it was added. Consider the following example: say we had $[1, 2, a]$, $[3, 4, b]$, and $[5, 6, c]$ in the tree. If we read an interval $[0, 7, d]$ where a, b, c are all left-of d, our tree will contain $[0, 1, d]$, $[1, 2, a]$, $[2.3.d]$, $[3, 4, b]$, $[4, 5, d]$, $[5, 6, c]$, and $[6, 7, d]$.

However, it turns out that the total number of intervals cannot double with every insertion. We will see this by counting not just intervals, but also ***gaps***. Gaps are memory ranges between consecutive intervals — in our example before d is inserted, $[0, 1]$, $[2, 3]$, $[4, 5]$, $[5, -]$ are gaps. When we insert an interval that doesn't overlap with any existing interval, we increase the number of intervals by exactly one and we increase the number of gaps by at most one, for a collective increase of at most 2 (we may not increase the number of gaps if the new interval is right next to another or decrease the number of gaps by one if we fill in the gap between two intervals). When we insert an interval that overlaps other intervals, we may increase the number of intervals by a lot, but only by filling in gaps. Therefore, the collective increase in the number of gaps and intervals is at most two in all cases. An empty tree has one gap. Since each insert increases the number of gaps and intervals (collectively) by at most 2, the total number of intervals is at most $2m + 1$. □

C Omitted Data in Section 5

Compile-Time vs. Runtime Coalescing. To get a better sense of how much the compile-time versus runtime coalescing can do, we separately collected various memory access pattern generated by running vanilla, compiler (compile-time coalescing) and by comp+rts (both compile-time and runtime), shown in Figure 9. First, we shall examine the numbers shown on the left side of the table: the numbers of accesses / intervals generated by all three version. In some benchmarks, such as `mmul` and `heat`, the compiler was able to coalesce in a non-negligible way, but in most benchmarks, the compiler cannot coalesce as much. The runtime coalescing other the other hand, seems much more effective in coalescing and deduplicating, leading to two or three order of magnitude of decrease in the number of intervals. Moreover, the average sizes of intervals (*avg.*) tend to be a lot larger with runtime coalescing.

Another question is, how much impact does the runtime deduplication have in reducing the overhead. We can gauge the answer to this question by looking at the total bytes that made into the access history (*sum*), also shown in the table. If the runtime performs coalescing only but not deduplication, the total bytes accessed should not change from compiler to comp+rts. Thus, by comparing the total bytes accessed generated by compiler versus comp+rts, we can tell that most benchmarks benefit from deduplication.

Copyright © 2022 by SIAM
Unauthorized reproduction of this article is prohibited

This data, combining with the data in Figure 5, indicate that while both compile-time and runtime coalescing can be beneficial, the benefit from runtime is more significant.

	hashmap	*treap*
`chol`	8.93	1.41
`fft`	207.72	392.50
`heat`	123.63	2.43
`mmul`	15.94	17.51
`sort`	26.36	1.54
`stra`	59.60	1.62
`straz`	52.00	3.50

Figure 10: The total time (in seconds) each benchmark spent updating its access history using hashmap (measured with comp+rts) versus treap.

Hashmap vs. Treap. Now we analyze the overhead of the treap construction in more detail and also explain why `fft` sees an overhead increase from comp+rts to treap. We measured the time that comp+rts and treap spent updating its respective access history. Indeed, the treap overhead is much larger than that of the hashmap in `fft`. Figure 10 shows the results.

Copyright © 2022 by SIAM
Unauthorized reproduction of this article is prohibited