Leveraging the Dynamic Program Structure Tree to Detect Data Races in OpenMP Programs

Lechen Yu
College of Computing
Georgia Institute of Technology
Atlanta, USA
lechen.yu@gatech.edu

Feiyang Jin
College of Computing
Georgia Institute of Technology
Atlanta, USA
fjin35@gatech.edu

Joachim Protze
IT Center
RWTH Aachen University
Aachen, Germany
protze@itc.rwth-aachen.de

Vivek Sarkar
College of Computing
Georgia Institute of Technology
Atlanta, USA
vsarkar@gatech.edu

***Abstract*—OpenMP provides a rich set of constructs to support multiple paradigms of parallelization, e.g., single program multiple data (SPMD) and task parallelism. Integrating these disparate paradigms in a single execution model increases the complexity of OpenMP, making OpenMP programs prone to concurrency bugs such as data races. Inspired by the task-oriented execution model in the OpenMP spec, we extended SPD3, a data race detection algorithm designed for async-finish task parallelism to support OpenMP programs. We found that by extending SPD3's key data structure, the Dynamic Program Structure Tree (DPST), SPD3 can support the majority of OpenMP constructs. We have implemented a prototype, TSan-spd3, on top of Google's ThreadSanitizer (TSan). To conduct an apples-to-apples comparison with Archer, the state-of-the-art dynamic race detector for OpenMP programs, we compared TSan-spd3 with an Archer implementation that executes on the same version of TSan. In addition, we evaluated Archer in two modes, the default mode using the original TSan and the accelerated mode enabling the use of SIMD instructions in TSan. The evaluation was conducted on nine benchmarks from the BOTS and SPEC OMP2012 benchmark suites. The evaluation results show that in eight out of nine benchmarks TSan-spd3 achieved similar overhead with Archer, while TSan-spd3 can identify more potential races than Archer.**

***Index Terms*—data race, OpenMP, performance**

I. Introduction

OpenMP is a popular parallel programming framework that has been widely adopted by HPC application developers [1]. With a rich set of provided constructs, OpenMP allows programmers to utilize multiple forms of parallelization, e.g., single program multiple data (SPMD) parallelism and task parallelism, in a single application. Integrating disparate parallel paradigms into OpenMP contributes to generality, but also increases the difficulty of correctly understanding OpenMP constructs. Incorrect usage of these constructs may incur *data races* during the program execution. As a common concurrency bug, a data race arises when two concurrent memory accesses operate on the same memory location and at least one of them is a write. Data races are notoriously pernicious because they may render the program output *nondeterministic*: under the same input, a parallel program may return inconsistent results in different runs. In addition, the indeterminacy of the program execution exacerbates the difficulty of detecting and debugging data races. Prior studies reveal that manually debugging data races is burdensome and time-consuming, even for experienced programmers [2].

In this paper, we focus on dynamic race detectors, i.e., correctness tools that execute along with parallel programs and accurately report encountered data races at runtime. Although data races have been well studied by the research community in past decades [3]–[5], only a handful of dynamic race detectors are available for OpenMP programs in practice. Archer is the state-of-the-art OpenMP-aware race detector [6]. It extends the functionality of Google ThreadSanitizer [7], known as TSan, by using TSan's annotation APIs. Upon intercepting the invocation of an OpenMP construct with the OpenMP Tool interface (OMPT) [8], Archer analyzes the construct's semantics and feeds the corresponding happens-before relation into TSan using annotation APIs. Archer fully leverages TSan's infrastructure, such as the instrumentation pass, vector-clock storage, and shadow memory, all of which are heavily optimized for dynamic race detection. Therefore, Archer can incur a comparatively low overhead to the program execution while exhibiting a low false positive rate.

The primary deficiency of Archer is the probability of missing data races (also called false negatives). Since TSan is designed for multithreaded programs, it does not take task parallelism into account. TSan uses vector clocks (VCs) [9] to compare the order of memory accesses, and all memory accesses from the same thread are always executed sequentially. Consequently, when two parallel OpenMP tasks are scheduled on the same thread, TSan cannot identify any data races between the two tasks. Since Archer reuses TSan's infrastructure, it also inherits this deficiency.

The OpenMP specification introduces a task-oriented execution model to describe the behavior of all supported parallelism. SPMD parallelism, task parallelism, and heterogeneous parallelism are all described using the notion 'task' (see Section II for more details)[1]. Inspired by this execution model, we studied SPD3, a precise race detection algorithm proposed for async-finish task parallelism [10]. We found that by extending its crucial data structure, Dynamic Program Structure Tree (DPST), SPD3 can tackle the majority of OpenMP constructs that are commonly used to implement SPMD parallelism and task parallelism in OpenMP programs. Furthermore, SPD3 only

[1]OpenMP defines task as an instance of executable code and its data environment that can be scheduled for execution on OpenMP by threads.

records a fixed number of historical memory accesses for each memory location, which is compatible with TSan's shadow memory implementation while incurring no loss of accuracy and completeness in data race detection (in particular, no false negatives).

We have built a prototype, TSan-spd3, to compare its performance with Archer. TSan-spd3 is developed on top of TSan. We reused TSan's instrumentation pass and shadow memory, and replaced the vector-clock-based race detection algorithm with SPD3. Like Archer, we also use OMPT to capture the invocation of OpenMP constructs and encode the corresponding happens-before relation into SPD3's DPST for data race detection. We picked up the base TSan implementation from LLVM 15, which has also integrated the latest implementation of Archer into its code base. Since both TSan-spd3 and Archer execute on the same version of TSan, we are able to conduct an apples-to-apples comparison of these tools. We have conducted a performance evaluation using nine benchmarks from BOTS and SPEC OMP2012, and evaluated Archer in two different modes - the default mode using the original TSan and the accelerated mode enabling the use of SIMD instructions in TSan. The evaluation results show that in eight out of nine benchmarks, TSan-spd3 achieved a similar time overhead with Archer in both modes.

Some prior work also builds on OpenMP's task-oriented execution model and encodes the happens-before relation into a tree or graph-based data structure to avoid missing data races [11]–[14]. However, prior work either uses different race detection algorithms or does not provide a detailed description of their prototypes' implementations. To the best of our knowledge, this work is the first to implement SPD3 in TSan and to make a fair comparison with Archer.

To summarize, the contributions of this paper are:

- An extension of SPD3 for OpenMP. We extend the original algorithm to support the semantics of OpenMP constructs.
- A complete evaluation comparing the performance of Archer and TSan-spd3 under different numbers of threads.
- An exploration of the prospect of graph-based race detection algorithms for tasks in OpenMP.
- TSan-spd3 is publicly available at https://github.com/lechenyu/TSan-spd3.

The rest of the paper is organized as follows: Section II introduces the task-oriented execution model in OpenMP spec and describes the details of SPD3. Section III analyzes Archer's implementation and explains the root causes of missing data races. In Section IV, we introduce our extension on SPD3 to support OpenMP constructs, followed by TSan-spd3's implementation in Section V. Section VI illustrates the evaluation results of TSan-spd3, as well as the comparison with Archer. Finally in Section VII we briefly conclude with a discussion of related work.

II. Background

In this section, we introduce OpenMP's task-oriented execution model. We also briefly describe the race detection algorithm used by SPD3.

A. OpenMP Execution Model

OpenMP introduces a unified execution model to delineate the behavior of all supported parallelization. An OpenMP program executes sequentially on a CPU until a `parallel` construct occurs. For each `parallel` construct, the underlying OpenMP runtime will assign several worker threads to parallelize the enclosed code, referred to as a *parallel region*. After all worker threads reach the end of the parallel region, the OpenMP program will resume sequential execution before encountering another `parallel` construct.

Task Parallelism. The OpenMP program begins as an *initial task* and is surrounded by an *implicit parallel region* containing only a single *initial thread*. During the execution, the initial task may explicitly fork other tasks using tasking constructs (e.g., `task` and `taskloop` constructs), and programmers can specify order constraints among tasks using synchronization constructs (e.g., `barrier` and `taskwait` constructs) or `depend` clauses. The execution model enforces that each OpenMP task is bound to the innermost enclosing parallel region, and can only be executed on those worker threads assigned to the parallel region. The actual task schedule on worker threads is determined by the OpenMP runtime, which is agnostic to programmers. By default, there is a barrier at the end of the parallel construct. All tasks created in the parallel region must terminate before the task invoking the `parallel` construct continues its execution.

SPMD. In OpenMP, SPMD can be achieved by combining a `parallel` construct with a `for` construct (a parallel-for loop). Figure 1 illustrates the behavior of such a loop using OpenMP tasks. When encountering a `parallel` construct, the OpenMP runtime will create a group of *implicit tasks* according to the number of requested parallelism. The name, implicit task, distinguishes it from other tasks explicitly created by tasking constructs. Implicit tasks execute the same parallel region on those worker threads assigned to the parallel construct. Upon reaching a `for` construct in the parallel region, the OpenMP runtime will distribute the iterations of the for-loop into implicit tasks. After the termination of all iterations, these implicit tasks will resume the redundant execution manner for the remaining part of the parallel region.

Heterogeneous Parallelism. OpenMP supports heterogeneous parallelism by device directives (e.g., the `target` construct). An OpenMP program can offload a code section to a connected accelerator to speed up the code section's execution. Such parallelization is also interpreted using tasks in the execution model. Since this paper focuses on data races on the CPU, the details of heterogeneous parallelism in OpenMP will be skipped.

B. SPD3 and DPST

Before introducing SPD3, We first explain a few fundamental notions in async-finish task parallelism. A parallel program applying async-finish task parallelism consists of two constructs, `async` and `finish`. Each "`async{s}`" statement creates a new child task with the body `s`, which will run asynchronously with the continuation of the parent task. Each "`finish{s}`"

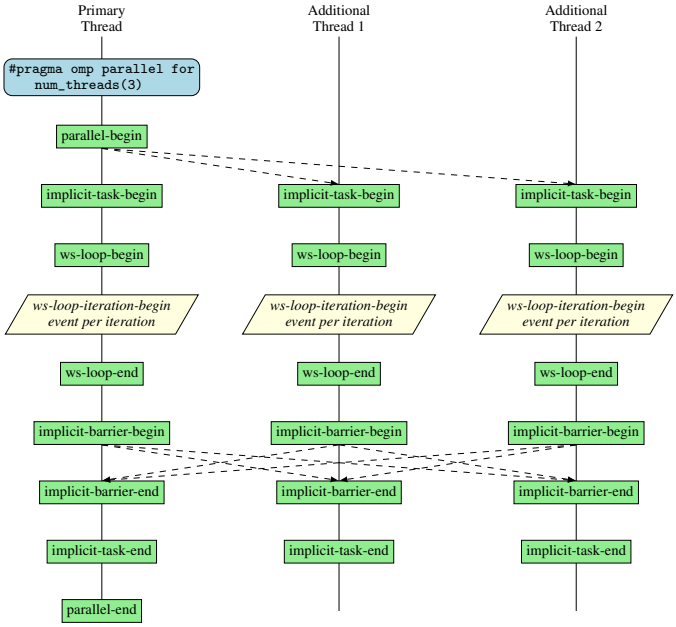

Fig. 1: OpenMP Events in a Parallel-For Loop

```
1  #pragma omp parallel num_threads(3){
2    S1
3    #pragma omp master
4    {
5      #pragma omp task {S2}
6      S3
7      #pragma omp taskwait
8      S4
9      #pragma omp taskgroup
10     {
11       S5
12       #pragma omp task {S6}
13       S7
14     }
15     S8
16   }
17   S9
18 }
```

Listing 1: An OpenMP Program Applying Task Parallelism

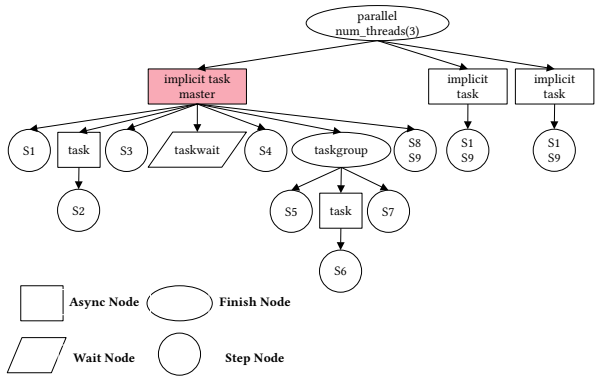

Fig. 2: DPST generated from Listing 1. The text in each node describes the associated OpenMP constructs or statements.

statement blocks the execution of the current task until all statements in the body `s` accomplish and all descendant tasks spawned in `s` terminate.

In OpenMP, tasking constructs can be treated as `async` statements, while `taskgroup` constructs and barriers are `finish` statements.

Happens-before check in SPD3. The key data structure of SPD3 is the Dynamic Program Structure Tree (DPST) which encodes the happens-before relation among tasks. Each leaf node in DPST corresponds to a *step* in the program. A step is a continuous region of code without any `async` or `finish` statement. On the other hand, each tree node in DPST corresponds to an `async` or a `finish` statement in the program. To check if a step node s_1 happens before another step node s_2, SPD3 first looks for the *lowest common ancestor* (LCA) of s_1, s_2. Among the children of LCA, there must be a node t such that t is an ancestor of s_1; s_1 happens before s_2 iff t is not an async node.

Figure 2 shows the corresponding DPST of the code snippet in Listing 1. Consecutive sibling step nodes are merged into a single node for memory efficiency. For example, the root's rightmost child has a merged step node representing $S1$ and $S9$. Let us take nodes $S1$ and $S7$ to illustrate the happens-before check. $LCA(S1, S7)$ is the red async node. The node t, in this case, is $S1$ itself, which is a step node, so that we can conclude that step $S1$ happens before step $S7$. On the other hand, for $S2$ and $S3$, their LCA is also the red async node, but this time the node t is S2's parent and it is an async node. Therefore, $S2$ and $S3$ may happen in parallel.

Shadow memory. For each memory location r, DPST saves the step nodes of the last write and at most two concurrent reads after the last write. These two concurrent reads are the leftmost and rightmost reads to r in DPST, such that other concurrent reads to r are within the subtree formed by the step nodes of the two reads and their LCA.

When a step s performs a write, SPD3 checks if the stored write and reads happen before s. If no race is detected, SPD3 will update the stored write and empty the two slots for reads.

When a step s performs a read, SPD3 checks if the stored write happens before s. If no race is detected, SPD3 will save it into one of the two read slots. If all slots are full, DPST will check if the new read can replace the leftmost or rightmost read and perform the replacement if the answer is yes; otherwise, the new read will be discarded.

The combination of the last write and two reads in each shadow memory location guarantees the precision and completeness of SPD3. With a given input, SPD3 can find out all data races after executing the program once, and it requires a fixed-size shadow memory to store historical memory accesses.

III. Discussion: Archer and its Correctness

Before presenting the experimental results, we discuss several design choices made by TSan, the underlying platform that Archer is built on. In short, these choices reduce Archer's time and memory overhead but sacrifice the completeness in data race detection.

A. How does TSan detect data races

TSan is a vector-clock-based race detector. A vector clock is essentially a vector of size n, where n equals the number of threads in the program. Each thread has a unique index, which indicates its entry position in the vector clock. When the program starts, each thread t will create a vector clock C_t in its local memory (so there are n vector clocks in total in the program), and all entries are initialized to zero. A thread updates values in the local vector clock when it performs a thread creation or synchronization event.

Happens-before check. Consider a thread t and its local vector clock C_t. For any other thread u, the clock entry $C_t(u)$ indicates the clock for the last operation of thread u that happens before the current operation of thread t.

Shadow memory. To improve the efficiency of locating shadow memory, TSan implements a fixed-size consecutive shadow memory for application memory sections. For each memory location, the start address of its *shadow cell* can be quickly retrieved by a handful of pointer arithmetic operations. By default, every 64 bytes of application memory is mapped to a shadow memory cell, and each cell has four slots to store memory accesses. A common case is that TSan saves the last write and three concurrent reads after the last write in a shadow cell. When the cell is full and a new concurrent read occurs, TSan will randomly clear a slot to store the new access. The information saved for each memory access is the thread id u and vector clock entry $C_u(u)$.

When a thread t performs a write, TSan checks if the stored write and reads happen before t. If no race is detected, TSan will update the stored write and empty the remaining three slots. When a thread t performs a read, TSan checks if the stored write happens before t. If no race is detected, TSan will save it to one of the three read slots.

B. Problem 1: missing races due to runtime scheduling

One key issue with Archer is that it only records synchronization events for each thread and barely considers other parallel information. For OpenMP, however, the relationship between tasks and threads is complex and needs extra care when carrying out data race detection.

A task in OpenMP packs some work to be done. A thread only executes a task if the runtime assigns the task to the thread. The challenge emerges when two concurrent tasks are assigned to the same thread at runtime.

Listing 2 presents an example from [15]. In the main function, the code snippet explicitly spawns three tasks which will be launched on the initial thread. The two tasks we are interested in here are the outer task created in line 6 and the inner task created in line 8. Inside these two tasks, line 10 and line 14 do not have a happens-before relation between them. These two lines can run asynchronously, and they both write to variable tp. In this example, it is safe to do so because tp is threadprivate (copied by value inside each thread). If we remove this constraint, there is a data race between line 10 and line 14.

```
1  int tp;
2  #pragma omp threadprivate(tp)
3  int var;
4
5  int main(){
6    #pragma omp task
7    {
8      #pragma omp task
9      {
10       tp = 1;
11       #pragma omp task {}
12       var = tp;
13     }
14     tp=2;
15   }
16
17   return 0;
18 }
```

Listing 2: Example from DataRaceBench (accessible on Github)

One valid execution trace for Listing 2 is lines 10, 12, and 14. When the execution comes to line 14, the thread performs a write, and Archer searches in the shadow memory to check which thread has accessed the variable. Archer will find that the previous write is also performed by this thread, thus reporting no race. In our experiment, Archer fails to find this race in some runs, which confirms our analysis.

Compared with Archer, SPD3 stores information on a task-based strategy. The inner and outer tasks will always be two distinct async nodes in DPST, disregarding the specific task schedule. In such a manner, SPD3 is guaranteed to report this race in all valid executions of this program. In our experiment, SPD3 always reports this race correctly, regardless of the number of threads and the task schedule. The complete result of the effectiveness comparison using DataRaceBench can be found at https://github.com/lechenyu/TSan-spd3.

C. Problem 2: missing races due to fixed-size shadow cell

For each memory location r, Archer only saves four most recent accesses, among which one is the last write and the rest are concurrent reads issued after the write. If a fourth read occurs, Archer will randomly eject one read from the slot to save the new read. This displacement mechanism may result in false negatives because the next write may be racy with the ejected read; in such case, Archer will miss the data race since the read was already removed from the shadow memory.

Saving all concurrent reads since the last write can fix this problem for Archer but hurt its performance. This fix may affect both the time and memory overhead.

1) Time overhead: at the time of a write, Archer will need to check races with at most n previous reads, where n is the number of concurrent reads since the last write. In the worst case, the time complexity of checking a write is linear to the number of threads.
2) Memory overhead: at present, Archer implements a fixed-size shadow memory. Archer limits each shadow cell to save four memory accesses, each with size x metadata (4 or 8 bytes). Assuming y memory locations are accessed

at runtime, the total size of shadow memory is $4 * x * y$. Nonetheless, once Archer removes the restriction of storing four accesses in each shadow cell, fixed-size shadow memory will no longer be feasible. The memory overhead, not surprisingly, will skyrocket.

In conclusion, if Archer decides to keep full access history instead of four accesses for each memory location, the time and memory overheads will escalate in scale. On the contrary, SPD3 only needs to save one write and two reads. The three stored memory accesses are sufficient to ensure precise race detection.

IV. SPD3 IN OPENMP

We now describe how we build SPD3 for OpenMP tasking features. As mentioned, SPD3 was initially designed for async-finish parallelism; some OpenMP constructs can be directly transformed while others require additional adaptation. Table I shows a summary of OpenMP constructs. The ✓ symbol means SPD3 can directly tackle this construct; the ✓* symbol means the construct is supported but needs modification to the original DPST construction; the × symbol means SPD3 cannot tackle the construct with the original happens-before check on DPST. We briefly explain each construct's expected behavior in OpenMP and how we model its happens-before relation using DPST:

- `parallel`: a parallel region generates certain number of tasks and has an implicit barrier at the end. This can be modeled as starting a new `finish` statement with a certain number of `async` statements in it.
- `task`: a `task` construct generates an explicit task. This can be modeled directly as an `async` statement.
- `taskwait`: a `taskwait` construct makes the current task wait for the completion of previously generated child tasks. The effect of `taskwait` is not recursive: the current task does not wait for descendants of child tasks. To handle `taskwait` in DPST, we insert a special *wait node* as a child of the current task. Each child keeps track of the preceding wait node. In the happens-before check, if there exists a wait node between the two step nodes, we can conclude that the node to the left of the wait node must happen before the node to the right; otherwise, the happens-before check should examine the LCA to determine the happens-before relation.
- `taskgroup`: a taskgroup construct makes the current task wait for the completion of its child tasks and their descendants. This can be modeled as starting a `finish` statement.
- `barrier`: a barrier indicates all threads in the parallel region should reach it before any thread can continue. We only consider the conditions that either all threads hit the barrier or none hit it. Under this assumption, a barrier can be treated as the end of the current parallel region and the start of a new parallel region.
- `masked/master`: these constructs specify that a subset of threads should execute the enclosed code section. Neither of them incur additional happens-before relation into the program.
- `single`: a `single` construct indicates only one thread should execute the enclosed code region, and other threads cannot proceed unless a `nowait` clause is specified. A `single` construct can be interpreted as a combination of `masked` and `barrier`.

TABLE I: Key OpenMP Constructs Supported by SPD3

OpenMP construct	Supported?
`parallel`	✓
`for`	✓
`task`	✓
`taskwait`	✓*
`taskgroup`	✓
`masked/master`	✓
`single`	✓*
`barrier`	✓*
`depend`	×
`critical`	×
device directives	×

Our tool currently does not support `depend`, `critical`, and other lock-based synchronization in OpenMP. The reason is that these constructs cannot be integrated into SPD3 without adding special edges that connect arbitrary nodes in the tree. To tackle `critical` and lock-based synchronization, an alternative race detection method is combining SPD3 with the Lockset algorithm [16]. Each task maintains a lockset to record acquired locks, and the lockset will be copied into the shadow cell when the task issues a memory access. During the happens-before check, if SPD3 returns that there exists no happens-before relation between the two steps in the DPST, the race detection method will additionally check the intersection of the two steps' locksets. If the intersection is not empty, then at least one common lock protects the accessed memory location, and thus the two steps are properly ordered. Otherwise, the two steps may happen in parallel, and the race detection method should report a data race.

Currently, TSan-spd3 does not support device directives since the memory space on the other device is not visible to the host. One exception is specifying the OpenMP offloading target to `x86_64-pc-linux-gnu`, which indicates that all target regions will be launched on the host, and data transfer will be simulated using `memcpy`. The `x86_64-pc-linux-gnu` target triple is designed to help programmers debug OpenMP applications utilizing device offloading. Since all memory regions reside in the same memory space on the host, TSan-spd3 can tackle such applications as regular CPU applications and identify data races in target regions.

Figure 2 illustrates several transformations discussed above. The top finish node corresponds to the parallel construct. Three threads execute the parallel region, so we insert three async nodes below the top finish. All three threads execute $S1$. Two worker threads do not execute the master region, so their next statement is $S9$. $S1$ and $S9$ can be combined into a single step node because no other OpenMP constructs exist between

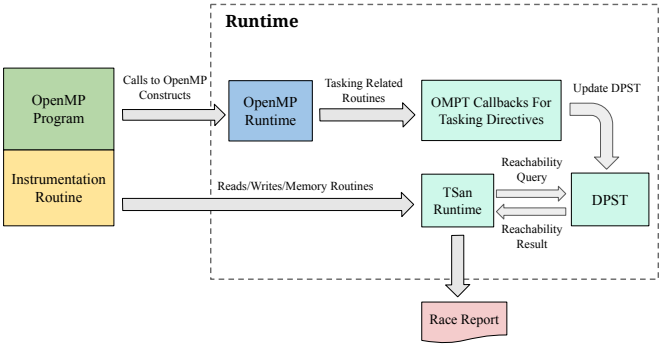

Fig. 3: TSan-spd3 Tool Flow

them.

The async node in red indicates the node executes in the master thread, and its children consist of all statements from line 2 to line 17. The wait node corresponds to the `taskwait` construct in line 7. It helps us to know that every step on its right-hand side ($S4$, $S5$, $S6$, $S7$, and the node associated with $S8$, $S9$) must wait for the completion of $S2$, while $S3$ can still run in parallel with $S2$.

V. Implementation

TSan-spd3 reuses TSan's instrumentation pass and shadow memory, while replacing TSan's vector clock implementation with DPST (see Figure 3 for TSan-spd3's architecture). Like Archer, TSan-spd3 also utilizes OMPT [8] to intercept invocations of OpenMP constructs, and OMPT provides an extensive interface for registering callbacks to OpenMP events. Upon an event of interest, our OMPT library will trigger the correlated callback to update DPST in TSan.

To correlate DPST with the stored memory accesses in the shadow memory, we add a field to the thread-local ThreadState struct that TSan maintains. The field saves a task's step node which is executing in the current thread. When a thread accesses a memory location, TSan-spd3 obtains the current step node and saves the information into the shadow cell.

When modifying the encoding of shadow cells for TSan-spd3, our first attempt was to save the step node pointer (64 bits) directly to each slot in the shadow memory. However, the new TSan implementation using SIMD instructions only assigns 32 bits for each slot in the shadow cell (22 bits are dedicated to the vector clock epoch). Considering the 32-bit size is hard-coded and cannot be easily changed, we keep the size unchanged but instead give each step node a 30-bit unique index. As a result, the step index of memory access can fit into a 32-bit slot. The remaining 2 bits in the slot are used to mark atomic operation and memory deallocation. In addition, each index is also the key when retrieving the corresponding step node. TSan-spd3 uses a lock-free global concurrency vector to store step nodes, which is created at the beginning of TSan-spd3's execution. A step index records the corresponding node's position in the concurrency vector so that the step node can be located in constant time.

VI. Evaluation

A. Experiment Setup

We conducted the evaluation on a single-node AMD server. It has two 12-core 3.8 GHz Ryzen9 3900X processors and 32 GB memory, running Ubuntu 18.04.3 LTS. TSan-spd3, Archer and all benchmarks are compiled by Clang/LLVM 15 using -O3 optimization.

For Archer, we evaluated its performance in two modes. The default mode invokes the original TSan, which uses a for-loop to iterate all four slots in the shadow cell and carry out vector clock comparison sequentially. The accelerated mode leverages a new implementation of TSan in LLVM 15. It parallelizes the vector clock comparison and shadow cell load/store using 128-bit SIMD instructions from the SSE2 instruction set. In addition, the new TSan implementation also compresses the size of each slot from 64 bits to 32 bits to load/store the whole shadow cell with a single SIMD instruction.

B. Benchmarks

For the evaluation, we picked up nine benchmarks from two open-sourced benchmark suites: BOTS 1.1 [17] and SPEC OMP2012 1.1 [18]. For BOTS, we examined each benchmark in the folder "omp-tasks", and there were seven benchmarks we could compile and run without any errors by Clang/LLVM 15 (fft, fib, health, nqueens, sort, strassen, uts). In SPEC OMP2012, there are three benchmarks using OpenMP tasking constructs: botsalgn, botsspar and kdtree. However, the input set provided for kdtree is too small. Even with the ref input set, kdtree's execution time is less than one second. To avoid bias from the underlying operating system, we decided to skip kdtree in the evaluation. Furthermore, the other two benchmarks, botsalgn and botsspar, are essentially ported from BOTS using OpenMP.

In the following paragraph, we briefly describe the functionality of the nine benchmarks. In addition, we also list the structure information of each benchmark in Table II. Column 'H' indicates the height of DPST, and the following four columns show the number of wait nodes, finish nodes, async nodes, and step nodes in the constructed DPST.

- **fft:** computes a Fast Fourier Transformation of a 6,000,000*6,000,000 matrix.
- **fib:** computes the 35th Fibonacci number.
- **health:** simulates the Colombian health care system; we estimate the running time for the medium model input file given in the source.
- **nqueens:** finds solution to the n-queens problem of input 13.
- **sort:** sorts an array of size 200,000,000 by a mixture of sorting algorithms.
- **strassen:** multiplies two matrices of size 4096 * 4096 using the Strassen algorithm with block size 64.
- **uts:** conducts an unbalanced tree search and we use the provided test.input as its input.
- **botsalgn:** aligns all protein sequences from an input file against every other sequence using the Myers and

Bench	H	Wait Node	Finish Node	Async Node	Step Node
fft	22	19,604,368	2	28,808,579	77,221,530
fib	37	14,930,351	1	29,860,723	74,651,800
health	7	17,515,985	1	17,515,626	52,547,240
nqueens	16	4,601,178	1	59,815,326	124,231,831
sort	20	1,157,801	1	2,490,380	6,138,563
strassen	10	19,608	1	137,279	294,169
uts	1,576	4,112,897	1	4,112,920	12,338,741
botsalgn	4	0	1	2,312	4,626
botsspar	4	300	1	292,547	585,398

TABLE II: Program Structure Information

Miller algorithm. We use the training size "100 sequences" provided in the benchmark.

- **botsspar:** computes the LU decomposition of a sparse matrix; the parameters we select for matrix size and submatrix size are 150 and 50.

C. Performance Comparison - Execution Time

In total, we conducted three groups of experiments using 8, 16, and 24 threads. We set the number of threads by the environment variable `OMP_NUM_THREADS`. We executed the nine benchmarks for each tool and thread number setting five times, then calculated the arithmetic mean of execution time and memory usage.

The execution time with 8, 16, and 24 threads are illustrated in Figure 4, Figure 5, and Figure 6, respectively. The vertical axis uses a base-10 log scale. "Base" denotes the original execution time, "Archer-simd" and "Archer" denotes the execution time of Archer with and without SIMD acceleration, and "TSan-spd3" denotes the execution time of our prototype. With the increase in the number of threads, the execution time reduces in most benchmarks, but the slowdown incurred by data race detection may increase. We observed that TSan-spd3 achieved similar performance with Archer in the majority of the nine benchmarks. In strassen, TSan-spd3 even used less time than Archer to accomplish data race detection. However, we also observed that in uts, TSan-spd3 is much slower than Archer. TSan-spd3 incurred a 100× slowdown, while Archer's slowdown was less than 10×. Table II shows that the DPST constructed for uts has an unusual height of 1576. Since the height of DPST dominates the time complexity of SPD3, it is reasonable that TSan-spd3 resulted in a higher slowdown than Archer. This example demonstrates that SPD3 may not perform well on all OpenMP programs. In the future, we may try to apply other techniques, e.g., static analysis, to further reduce the overhead of SPD3 for those OpenMP programs which construct a deep DPST.

D. Performance Comparison - Memory Usage

Figure 7 illustrates the memory usage with 24 threads. We omit the results with 8 and 16 threads because we found the number of threads has little effect on the memory usage of

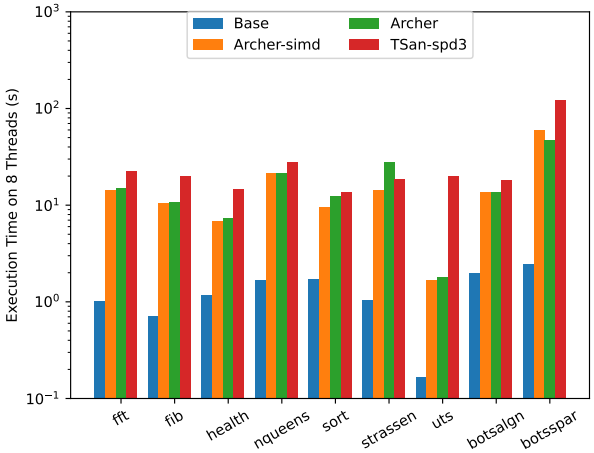

Fig. 4: Execution Time Using 8 Threads

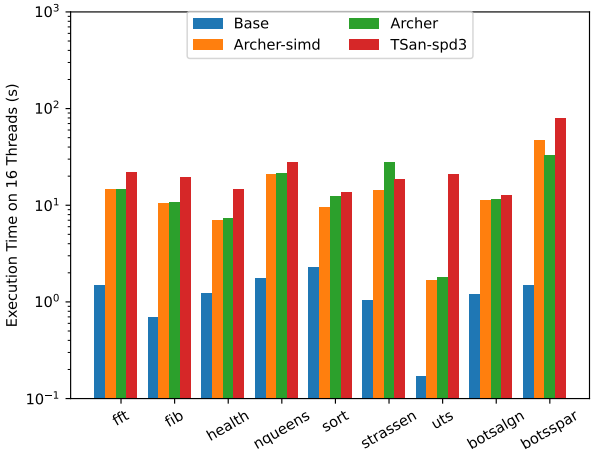

Fig. 5: Execution Time Using 16 Threads

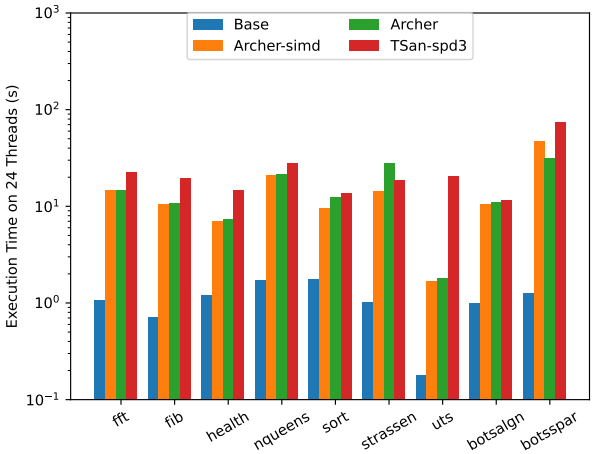

Fig. 6: Execution Time Using 24 Threads

Bench	DPST Size (GB)
fft	8.42
fib	8.01
health	5.87
nqueens	12.64
sort	0.65
strassen	0.03
uts	1.37
botsalgn	0.0004
botsspar	0.05

TABLE III: DPST Memory Usage

TSan-spd3 and Archer. As a prototype, we have not optimized the storage of DPST for TSan-spd3. So we expect that the shadow memory usage shall be close to Archer, while the DPST may occupy a large amount of memory space.

The results in Figure 7 reveal that TSan-spd3 requested much more memory than Archer. Combined Figure 7 with the statistics on DPST sizes in Table III, we can conclude that the majority of TSan-spd3's memory usage results from the DPST or the preallocated concurrency vector to store DPST's step nodes. TSan-spd3 allocates a large concurrency vector to store DPST's step nodes before running the OpenMP programs. To ensure the concurrency vector has enough space for all nine benchmarks, we currently set the size of the concurrency vector to 200,000,000. Because each step node requires 72 bytes in memory space, the overall size of the concurrency vector is 13.41 GB. For benchmarks creating a small DPST such as sort, strassen, spar, and algn, the overall memory usage minuses the concurrency vector size is similar to or even smaller than Archer's memory usage. For benchmarks creating a large DPST such as fib, fft, nqueens, the memory usage is dominated by DPST and the concurrency vector, in which the shadow memory only occupies a small portion of the overall memory usage.

There are a few ways to reduce the memory usage of TSan-spd3. First, the concurrency vector's size is configurable. Programmers can set it to a reasonable size according to an estimate of DPST size. Such an estimate can be easily acquired by an OMPT tool which counts the number of tasking and synchronization constructs during the execution. We will also start optimizing TSan-spd3's memory usage by removing unnecessary fields in the tree nodes and step nodes of DPST.

VII. Related Work

Race detection for task parallelism has been well studied. Labeling techniques have also been used in past work on race detection for task parallelism. This approach enables happens-before check between two nodes by comparing two labels. Mellor-Crummey introduced the Offset-Span [19] algorithm as one such approach, in which the length of the label attached to each task can grow as large as the depth of nested fork structures. The SP-Bags [20] structure devised by Feng and Leiserson, and the ESP-Bags [21] introduced by Raman et al. are also examples of using labeling to record the happens-before relation for a task-parallel program.

There has been a long history of dynamic race detection algorithms and tools based on vector clocks [7], [22], [23]. A major advantage of the vector clock approach is that it can be applied to parallel programs with arbitrary parallel constructs, including locks and transactions. However, its major limitation when applied to task-parallel programs is that it can only provide guarantees on a per-schedule (rather than per-input) basis since it is not practical for vector clock sizes to be proportional to the number of active tasks.

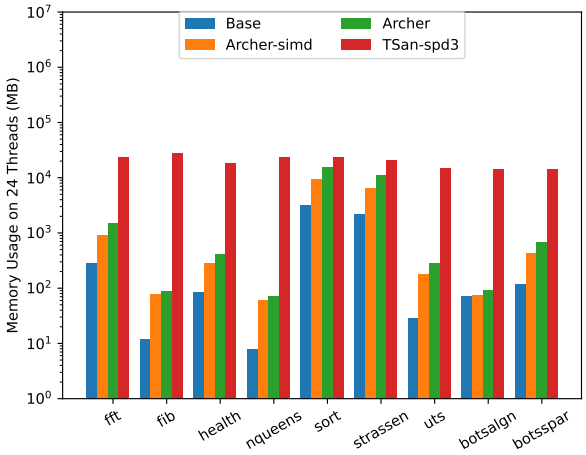

Fig. 7: Memory Usage with 24 Threads

VIII. Conclusion

In this paper, we extend SPD3 to support OpenMP programs. We built a prototype TSan-spd3 on top of TSan and conducted an apples-to-apples comparison with Archer. The evaluation on nine open-sourced benchmarks shows that TSan-spd3 achieved a similar overhead to the program execution compared to Archer, while avoiding missing data races at runtime.

For future research, we will take depend clause into account. We plan to extend SPD3 further to support point-to-point synchronization in OpenMP programs.

References

[1] OpenMP ARB, "Who's Using OpenMP," https://www.openmp.org/about/whos-using-openmp/, 2019, accessed: August 12, 2022.

[2] K. Sato, D. H. Ahn, I. Laguna, G. L. Lee, M. Schulz, and C. M. Chambreau, "Noise injection techniques to expose subtle and unintended message races," in *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2017, pp. 89–101.

[3] R. H. Netzer and B. P. Miller, "What are race conditions? some issues and formalizations," *ACM Letters on Programming Languages and Systems (LOPLAS)*, vol. 1, no. 1, pp. 74–88, 1992.

[4] U. Banerjee, B. Bliss, Z. Ma, and P. Petersen, "A theory of data race detection," in *Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging*, 2006, pp. 69–78.

[5] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, 2008, pp. 329–339.

[6] S. Atzeni, G. Gopalakrishnan, Z. Rakamaric, D. H. Ahn, I. Laguna, M. Schulz, G. L. Lee, J. Protze, and M. S. Müller, "Archer: Effectively spotting data races in large openmp applications," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 53–62.

[7] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: Data race detection in practice," in *Proceedings of the Workshop on Binary Instrumentation and Applications*, ser. WBIA '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 62–71. [Online]. Available: https://doi.org/10.1145/1791194.1791203

[8] A. E. Eichenberger, J. Mellor-Crummey, M. Schulz, M. Wong, N. Copty, R. Dietrich, X. Liu, E. Loh, and D. Lorenz, "Ompt: An openmp tools application programming interface for performance analysis," in *OpenMP in the Era of Low Power Devices and Accelerators*, A. P. Rendell, B. M. Chapman, and M. S. Müller, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 171–185.

[9] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 179–196.

[10] 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*, ser. PLDI '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 531–542. [Online]. Available: https://doi.org/10.1145/2254064.2254127

[11] Y. Gu and J. Mellor-Crummey, "Dynamic data race detection for openmp programs," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 767–778.

[12] N. Boushehrinejadmoradi, A. Yoga, and S. Nagarakatte, "On-the-fly data race detection with the enhanced openmp series-parallel graph," in *International Workshop on OpenMP*. Springer, 2020, pp. 149–164.

[13] H. S. Matar and D. Unat, "Runtime determinacy race detection for openmp tasks," in *European Conference on Parallel Processing*. Springer, 2018, pp. 31–45.

[14] S. Atzeni, G. Gopalakrishnan, Z. Rakamaric, I. Laguna, G. L. Lee, and D. H. Ahn, "Sword: A bounded memory-overhead detector of openmp data races in production runs," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, pp. 845–854.

[15] C. Liao, P.-H. Lin, J. Asplund, M. Schordan, and I. Karlin, "Dataracebench: A benchmark suite for systematic evaluation of data race detection tools," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3126908.3126958

[16] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Transactions on Computer Systems (TOCS)*, vol. 15, no. 4, pp. 391–411, 1997.

[17] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, "Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp," in *Proceedings of the 2009 International Conference on Parallel Processing*, ser. ICPP '09. USA: IEEE Computer Society, 2009, p. 124–131. [Online]. Available: https://doi.org/10.1109/ICPP.2009.64

[18] "SPEC OMP® 2012," https://www.spec.org/omp2012/, 2016.

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

[20] M. Feng and C. E. Leiserson, "Efficient detection of determinacy races in cilk programs," in *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 1–11. [Online]. Available: https://doi.org/10.1145/258492.258493

[21] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav, "Efficient data race detection for async-finish parallelism," *Formal Methods in System Design*, vol. 41, no. 3, pp. 321–347, Dec 2012. [Online]. Available: https://doi.org/10.1007/s10703-012-0143-7

[22] C. Flanagan and S. N. Freund, "Fasttrack: efficient and precise dynamic race detection," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 121–133, 2009.

[23] Intel, "Intel inspector," May 2021. [Online]. Available: https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/inspector.html#gs.1wvmbu