SWORD: A Bounded Memory-Overhead Detector of OpenMP Data Races in Production Runs

Simone Atzeni, Ganesh Gopalakrishnan, Zvonimir Rakamarić
University of Utah
Salt Lake City, UT, United States
{simone, ganesh, zvonimir}@cs.utah.edu

Ignacio Laguna, Gregory L. Lee, Dong H. Ahn
Lawrence Livermore National Laboratory
Livermore, CA, United States
{ilaguna1, lee218, ahn1}@llnl.gov

***Abstract*—The detection and elimination of data races in large-scale OpenMP programs is of critical importance. Unfortunately, today's state-of-the-art OpenMP race checkers suffer from high memory overheads and/or miss races. In this paper, we present SWORD, a data race detector that significantly improves upon these limitations. SWORD limits the application slowdown and memory usage by utilizing only a *bounded, user-adjustable* memory buffer to collect targeted memory accesses. When the buffer fills up, the accesses are compressed and flushed to a file system for later offline analysis. SWORD builds on an operational semantics that formally captures the notion of concurrent accesses within OpenMP regions. An offline race checker that is driven by these semantic rules allows SWORD to improve upon happens-before techniques that are known to mask races. To make its offline analysis highly efficient and scalable, SWORD employs effective self-balancing interval-tree-based algorithms. Our experimental results demonstrate that SWORD is capable of detecting races even within programs that use over 90% of the memory on each compute node. Further, our evaluation shows that it matches or exceeds the best available dynamic OpenMP race checker in detection capability while remaining efficient in execution time.**

***Index Terms*—Dynamic Data Race Detection; Concurrency Bugs; Data Races; OpenMP; High Performance Computing; HPC; Offline Analysis**

I. INTRODUCTION

Given the inexorable march toward higher computational efficiencies, many critical software components are being transitioned to adopt on-node parallelism. The predominant parallel programming model of choice in this endeavor is OpenMP. Even though OpenMP provides constructs that ease the expression of parallelism, programmers still introduce data races that may appear innocuous at first glance, but in fact have serious consequences (an example involving the Hypre library is provided in related work [1]). Such incidents and recent studies (e.g., [2]) have helped elevate the importance of data race checking of large-scale OpenMP programs. Static-analysis-based data race detection tools are often considered by those aiming for scalability; however, these tools are also known for their high false alarm rates [3], [4], [5], [6]. As a result, dynamic analysis is preferred, with four recent OpenMP race checking tools based on it being Helgrind [7], TSan [8], [9], Intel®Inspector XE [10], and ARCHER [1]. Recent work [2] provides a comparative study of these tools on the also contributed data race benchmark suite. Overall, while Helgrind and TSan are well-engineered and mature tools, they are fundamentally designed for low-level models such as POSIX Threads. Since they do not model OpenMP synchronization, they end up generating false alarms on real OpenMP programs.

ARCHER has emerged as a tool capable of handling realistic programs in production settings and avoiding false alarms, thanks to the incorporation of the OpenMP synchronization semantics [11]. In addition, ARCHER owes its success to the use of a static analysis phase that excludes statically-guaranteed race-free loops from dynamic analysis, and its reliance on the TSan engine—a well-engineered implementation of happens-before race checking. However, ARCHER suffers from three significant drawbacks: high memory overhead, race omission due to shadow-cell evictions, and happens-before race masking. In this paper, we introduce a fully redesigned new race checker called SWORD that overcomes these limitations. We now describe these drawbacks and point out how SWORD overcomes them.

a) High Memory Overhead: Happens-before race checkers typically log read and write accesses, assigning them logical time instances (e.g., vector clock values or epochs [12]). Ideally, such tools must maintain all memory accesses. Unfortunately, this is impossible in practice, given the large number of variables and accesses in realistic programs. As a compromise, TSan, and hence also ARCHER, only maintain *four*[1] memory accesses per 8 bytes of application memory (hereafter called a *memory word*). Each access record (called a *shadow cell*) also occupies one word. Thus, it is clear that the memory consumption *quintuples* (and in practice, it goes up 6-fold due to other per-thread overhead). We have observed this when ARCHER was applied on the AMG2013 benchmark: the 6-fold increase with respect to total application memory gave us an out-of-memory (OOM) error. There is no easy way to predict application memory needs, and thus OOM is a lurking danger even with only four shadow cells.

SWORD has very low memory requirements, as it does

This work was performed under the auspices of the U.S. Department of Energy by LLNL under Contract DE-AC52-07NA27344 (LLNL-CONF-740324), NSF OAC 1535032, and NSF CCF 1704715.

[1]A default setting, but adjustable between 1 and 8.

1530-2075/18/$31.00 ©2018 IEEE
DOI 10.1109/IPDPS.2018.00094

not employ shadow cells. In SWORD, each thread collects memory accesses into its own buffer, and it suffices to allocate a fixed *2 megabytes per thread* buffer.[2] When each buffer fills up, the associated thread (independent of the other threads) compresses and writes out the buffer to disk. The advantages of independently collecting the traces are several. For instance, when collecting events associated with OpenMP barriers, the threads do not have to wait for each other to finish the barrier.

b) Race Omission due to Shadow-Cell Evictions: Since ARCHER retains and analyzes only four accesses per memory word, a fifth access ends up evicting one of these cells. Unfortunately, this results in missed races, as has been observed while using ARCHER on real-world applications. SWORD does not suffer from this drawback, as it retains *all* accesses.

c) Happens-Before Race Masking: A happens-before race checker such as ARCHER can mask races when otherwise conflicting accesses are separated by a happens-before path created as an artifact of the particular schedule (see Figure 1). This form of race masking is reported in prior literature [13], [14], and also shows up in practice while using ARCHER.

In SWORD, an *offline* synchronization recovery and race analysis phase detects races. This phase is driven by an operational semantics of OpenMP [15] that determines which accesses are concurrent. This approach completely avoids happens-before race masking. It also directly supports independent trace collection; for instance, it is the offline analysis that helps us put together the separately collected OpenMP barriers. Overall, SWORD aims to guarantee completeness of data race checking for a given execution if it does not have data-dependent branches.

Highlights of SWORD*'s Implementation*

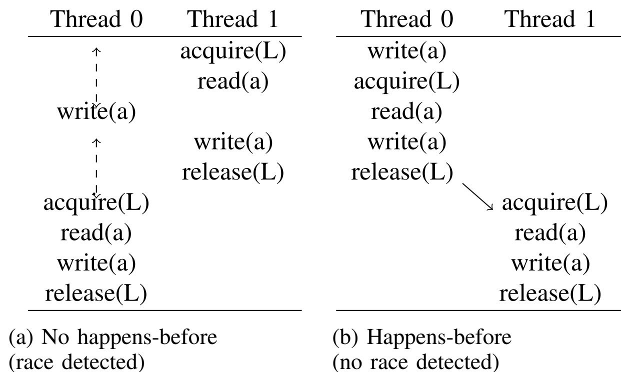
Our initial implementation of this approach in SWORD proved quite disappointing, as some examples took days to run. After careful optimization, we brought down this time for the same examples to a few seconds. Our use of the following mechanisms was central to achieving this performance:

- state-of-the-art self-balancing interval trees for recording and merging traces;
- an efficient realization of Offset-Span Labels [16] for concurrency discovery; and
- constraint solving to detect conflicting accesses through complex strided access patterns and partial word overlaps.

SWORD also enjoys high portability, thanks to its use of a standard trace collection method based on OMPT, an emerging tools interface that is expected to be incorporated into future OpenMP standards [17]. To summarize, the contributions of SWORD are as follows:

- bounded memory (as little as 2 MB) instead of taking gigabytes of shadow-cell storage;
- free of race omissions due to shadow-cell evictions;
- no happens-before-induced race masking;
- publicly available as an open-source GitHub project at https://github.com/PRUNERS/sword.

[2]A user-adjustable bound, but we found that 2 MB is typically optimal since it easily fits within modern L3 caches.



Thread 0	Thread 1
	acquire(L)
	read(a)
write(a)	
	write(a)
	release(L)
acquire(L)	
read(a)	
write(a)	
release(L)	

(a) No happens-before (race detected)

Thread 0	Thread 1
write(a)	
acquire(L)	
read(a)	
write(a)	
release(L)	
	acquire(L)
	read(a)
	write(a)
	release(L)

(b) Happens-before (no race detected)

Figure 1: Different interleavings generated by the same program. Dashed lines indicate that the write operations of Thread 0 can occur simultaneously with the operations of Thread 1. Solid lines indicate happens-before edges between the threads.

II. BACKGROUND

A data race occurs when two concurrent memory accesses (one of which is a write) target the same memory location. Dynamic race detectors employ *happens-before* (typically implemented using *vector clocks* [18] or variants) to determine whether two accesses are concurrent. Given a thread schedule (interleaving), the underlying concurrency semantics yields a happens-before relation. Figure 1 shows two possible interleavings of the same program. In part (a), a race is caught because of the absence of any happens-before ordering between Thread 0's write(a) invocation and Thread 1's read(a) or write(a) invocation. In part (b), write(a) of Thread 0 is happens-before ordered before both read(a) and write(a) accesses of Thread 1, causing the race to be missed. This is one common source of missed races we observe in ARCHER. Notice that even without any branches in the code, the choice of interleavings decides whether a race is detected or missed. In SWORD, this sort of race omission does not happen, as the true concurrency status of two accesses is computed using our operational semantic model.

To further detail shadow-cell eviction mentioned in the previous section, consider the following example harboring a race with respect to $a[0]$ because, while multiple threads read the array location $a[0]$, exactly one thread is arranged to write it without synchronization:

```
int a[N];

#pragma omp parallel for
for(int i = 0; i < N; i++) {
  a[i] = a[i] + a[0];
}
```

Suppose the master thread (thread 0) got a head start and created an access record of $a[0]$ being written. Given the multiple reads on $a[0]$ from other threads, it is possible that this write record is purged before race-checking is invoked (i.e., all 4 shadow cells hold read accesses). In this situation, this race can be missed.

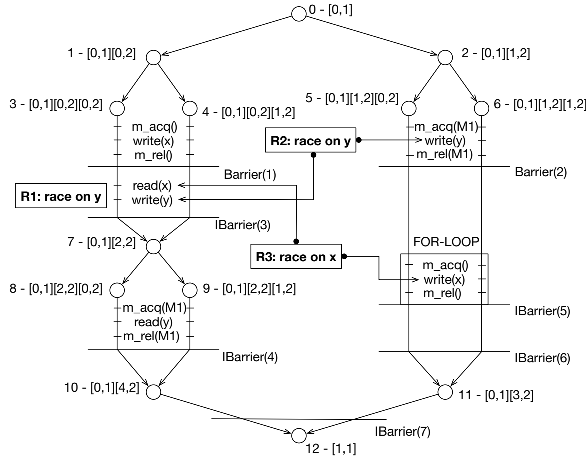


Figure 2: Structure of an OpenMP program, where m_acq/m_rel denotes a mutex acquire/release, $IBarrier(id)$ an implicit, and $Barrier(id)$ an explicit OpenMP barrier.

Concurrency Structure of OpenMP: Figure 2 shows the concurrency structure of an OpenMP program with two nested parallel regions where threads access shared memory locations. The figure depicts OpenMP barriers, as well as memory accesses and synchronization operations in-between. We define a *barrier interval* to be a pair of adjacent barriers along with the set of memory events spanned by them. For example, Barrier Interval 3 includes the operations performed between barriers 1 and 3 (in general, some of these barriers could be implicit barriers—denoted by *IBarrier*—such as introduced by default at the end of OpenMP parallel sections).

Our offline analysis phase associates each memory access event or synchronization event it receives with the barrier interval within which the event occurs. It is easy to observe that the accesses carried out by two different threads within the same barrier interval are concurrent and potentially can race. For example, data race R1 happens within the same Barrier Interval 3 between threads 3 and 4 since they both write to y without synchronization. However, accesses within sequentially ordered barrier intervals cannot race. For instance, the write to x in Barrier Interval 1 by Thread 3 cannot have a data race with the read of x by Thread 4 in Barrier Interval 3, as these accesses are separated by a barrier (and hence are sequentially ordered). However, with *nested* parallelism, two threads that belong to two different barrier intervals can in fact race; races R2 and R3 are of this nature. These two data races happen because the threads are accessing shared variables (y for R2 and x for R3) from two barrier intervals that belong to different concurrent parallel regions. Our offline analysis phase relies on *offset-span labels* to identify if two accesses are concurrent (we apply the data race analysis only to concurrent accesses).

Offset-Span Labels: An offset-span label tags each thread's execution point with a sequence of pairs (e.g., $[0,1][0,2][0,2]$), marking its lineage in the concurrency structure defined by prior forks and joins. By comparing these labels we can determine if two threads are concurrent, thereby focusing the data race analysis only to potentially racy threads.

The domain for the offset-span labels is $OSL = (\mathbb{N} \times \mathbb{N})^*$, i.e., each member $osl \in OSL$ is a sequence of pairs $[a_1, b_1][a_2, b_2], \ldots, [a_n, b_n]$. A pair consists of *offset* and *span*. The span indicates the number of threads spawned by the fork (e.g., start of a parallel region) from which the pair originates. The offset distinguishes the pair among the other pairs originating from the same parent. Take label $[0,1][0,2][0,2]$ of Thread 3 in Figure 2 as an example. Starting from the end, the pair $[0,2]$ indicates that the thread has ID 0 in a parallel region of two threads; the second pair $[0,2]$ is the thread's parent with ID 0 in a parallel region of two threads; the first pair $[0,1]$ is the predecessor of the thread's parent and represents the master thread.

Let $osl_1, osl_2 \in OSL$ be two offset-span labels associated with Thread 1 and Thread 2, respectively. These labels are sequential (i.e., Thread 1 and Thread 2 are not concurrent) when either

case 1: $\exists\, P, S \,.\, osl_1 = P \wedge osl_2 = PS$, where P and S are non-empty sequences of pairs, or

case 2: $\exists\, P, S_x, S_y, o_x, o_y, s \,.\, osl_1 = P[o_x, s]S_x \wedge osl_2 = P[o_y, s]S_y \wedge o_x < o_y \wedge o_x \bmod s = o_y \bmod s$, where P, S_x, S_y are (possibly empty) sequences of pairs.

Otherwise, the labels are concurrent (see [16] for details). This judgement of concurrency is not based on building happens-before, thus avoiding problems such as highlighted in Figure 1(b): in particular, SWORD will detect this race. However, our semantics (and hence SWORD's implementation) does not take into account data-dependent control flows. Thus, if a program bases its control-flow branch decisions on the order in which prior synchronization actions have been executed, then SWORD will miss races. Barring this, SWORD is a faithful realization of our semantics (albeit, coded manually).

III. Implementation Details

A. Dynamic Analysis

Compiler Instrumentation: We implemented SWORD using the LLVM/Clang tool infrastructure [19] (see Figure 3). Our LLVM instrumentation pass instruments all load and store instructions that are executed within a parallel region. (We ignore sequential instructions as they cannot race.)

Log Collection: At runtime, SWORD collects all the information necessary for offline data race detection. Recall that each thread gathers its logs without coordination with the other threads. For this, the threads interact with the OpenMP runtime through the OMPT interface, and gather all the information regarding thread creation, parallel region begin/end, and synchronizations points (e.g., barriers, critical section). OMPT provides a data field for each callback, and we generate unique IDs for each OpenMP construct analyzed (e.g., ID for a parallel region, ID for a critical section); we

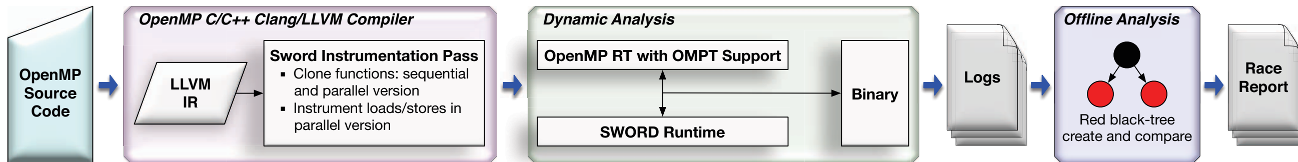


Figure 3: SWORD tool flow.

pid	ppid	bid	offset	span	level	data begin	size
0	–	0	0	24	1	0	50,000
0	–	1	0	24	1	50,000	75,000
1	–	0	0	24	1	75,000	10,000

TABLE I: Example of thread's meta-data file. Each line corresponds to one barrier interval. Column **pid** is parallel region ID, **ppid** is parent parallel region ID, **bid** is barrier ID, **offset** and **span** define offset-span label, **level** is level of parallelism, **data begin** is offset (in bytes) in the log file of the beginning of the respective data chunk, **size** is its size.

store this information inside the data field so that the IDs can be retrieved during the logging process and stored into log files. Meanwhile, the instrumented parallel loads and stores gather information about every parallel memory access (e.g., size, read or write, atomic).

Each thread maintains one log file and one meta-data file. The log file contains the information about memory accesses and OpenMP events, while the meta-data file contains the IDs of parallel regions, offsets into the log file to obtain the data (i.e., memory accesses and OpenMP events) regarding a specific parallel region, and other information. Table I details each thread's meta-data file, which helps the offline analysis identify the concurrency structure. Each line in the meta-data file represents a barrier interval. This information is used by the offline data race detection algorithm to extract from the log file the chunk of data for a specific barrier interval.

During program execution, SWORD collects the memory accesses and OpenMP events information into limited-size thread-local storage buffers. When a buffer gets full, it is compressed and asynchronously written out into a log file. We compared several open-source compression algorithms, namely *LZO* [20], *Snappy* [21], and *LZ4* [22]. In our case, they all have similar performance and compression ratios, and we chose *LZO* since it was easier to integrate into SWORD.

Bounded Dynamic Analysis Overhead: As previously mentioned, during the dynamic analysis each thread maintains a thread-local storage buffer to collect memory accesses and OpenMP events before writing them into a file. We fine-tuned the buffer size to minimize cache misses, and we found that an optimal size for our setup holds 25,000 events, amounting to around 2 MB total. The SWORD runtime maintains additional information in several thread-local storage variables. The amount of memory needed by SWORD for this auxiliary storage and OMPT is around 1.3 MB per thread. Given that the memory overhead is bounded and independent of the characteristics of the analyzed application, we can obtain a formula representing the total memory overhead of SWORD. Let N be the number of threads, B the memory overhead introduced by SWORD per thread, and C the memory overhead introduced by the OMPT interface. Then, the total memory overhead of SWORD is $N \times (B+C)$. Our experimental results show that in our setup the total memory overhead of SWORD is around 3.3 MB per thread (which includes the aforesaid auxiliary storage).

B. Offline Analysis

Offline analysis starts by analyzing the meta-data files to identify the concurrency structure. Once the algorithm has identified all pairs of concurrent barrier intervals and threads, it obtains information about the memory accesses and OpenMP synchronization operations from the log files. The meta-data file contains an offset for each barrier interval indicating the location of pertinent data in the log files. The size of a single log file can be *dozens of gigabytes*, and hence the entire data collection from an application can be in the order of terabytes. Thus, even without application memory pressure, it is not always possible to analyze all the data directly in memory. To handle large log files efficiently, we employ a *streaming algorithm* [23] that reads access information from log files in small chunks and carries out our analysis.

For each thread, the algorithm builds an *interval tree* to summarize memory accesses and to maintain information about OpenMP events. In our implementation, we use an augmented red-black tree [24] to maintain the interval tree balance and to speed up the operations of insertion and search. A node in an interval tree contains the range of memory accesses[3] it represents, and auxiliary information such as the operation type (R/W), size of the access, stride of the interval, program counter, and mutex set. The interval tree approach allows us to summarize the information about consecutive memory accesses (e.g., array accesses) in one node. Data race detection is then performed by comparing the interval tree of each thread to the interval trees of other concurrent threads. When a node in the tree overlaps with a node of another tree there is a *potential race*.

Figure 4 shows an example of two threads accessing an array of structures. Each thread is accessing a different field of the structure, performing either a read or write, and there are no overlapping accesses—hence also no data race. During the offline analysis, SWORD summarizes the accesses of both threads using the two shown intervals. The two intervals do overlap; *however, if we consider the size and the stride of the*

[3] We treat a single access as a range with the same beginning and end.

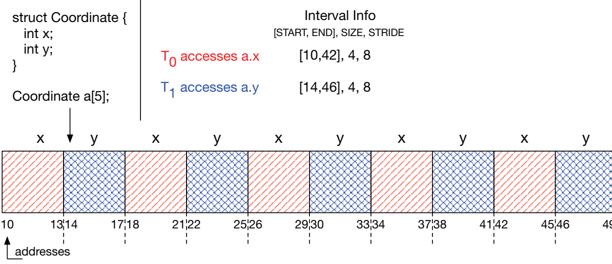


Figure 4: Example of threads that access the same memory interval but do not have common addresses.

accesses, they do not actually have any addresses in common, as the threads are accessing different memory locations. Thus, a simple overlap check is not sufficient to identify whether two intervals intersect.

In our offline race detection algorithm, we use all the available interval information (e.g., count, stride) to check if two intervals have memory addresses in common. For an interval of thread T_i, we represent all addresses that belong to it with the following constraint:

$$\begin{aligned} &\Delta \cdot x_i + b_i + s_i = a \\ &\wedge\ 0 \leq x_i \leq ((e-b)/\Delta) \\ &\wedge\ 0 \leq s_i < s, \end{aligned}$$

where a is an address belonging to the interval, b and e are the starting and ending address of the interval respectively, Δ is the stride, and s is the size of the memory access. If we consider the example in Figure 4, we can represent all the addresses for intervals of T_0 and T_1 with these constraints:

$$\begin{aligned} T_0: \quad & 8 \cdot x_0 + 10 + s_0 = a & T_1: \quad & 8 \cdot x_1 + 14 + s_1 = a \\ & \wedge\ 0 \leq x_0 \leq 4 & & \wedge\ 0 \leq x_1 \leq 4 \\ & \wedge\ 0 \leq s_0 < 4 & & \wedge\ 0 \leq s_1 < 4 \end{aligned}$$

If their conjunction is satisfiable, then the threads are accessing a common address. Furthermore, if at least one of the operations is a write, then a race is reported. In our implementation, we use integer linear programming to solve the constraints, and in particular GNU GLPK Version 3.63 (any other solver with similar capabilities could be employed).

The algorithm complexity is $O(Nlog(N))$ for the interval tree creation with N being the number of memory accesses: it takes $O(log(N))$ to insert a node into a tree and this is done for all N memory accesses. The comparison of two interval trees is $O(Mlog(M))$ with M being the number of nodes in the tree: each of the M nodes in a tree is compared to the other trees, which is a binary search with complexity $O(log(M))$. Note that $M \leq N$ since an interval tree can summarize multiple access into one interval node.

Interval Tree Example: The following example, when executed with two threads, contains a data race in array *a* due to a data dependency:

```
int a[1000];

#pragma omp parallel for num_threads(2)
for(int i = 1; i < 1000; i++) {
  a[i] = a[i - 1];
}
```

During the dynamic analysis, SWORD generates two log files and two meta-data files. Since the program has only one parallel region and one barrier interval, the meta-data files contain only one line. The offline data race detection algorithm extracts the barrier interval data using the meta-data files, and builds one red-black interval tree per thread.

Figure 5 shows possible interval trees for the two threads executed by the program. Each node in an interval tree describes a memory access or a collection of memory accesses (e.g., array accesses). In addition, each node has fields to store information about the operation type (read or write), size of the memory access, program counter, and list of mutexes held for that specific memory access. When the algorithm identifies two overlapping intervals, as shown in red/underlined in Figure 5, it employs the additional information in nodes to construct an integer linear constraint used to check if there is a potential race. The algorithm also checks whether one of the intervals is a write operation and if the intersection of the mutex lists is empty. If these two conditions are met and the linear constraint is satisfiable, a race is reported. In the case of Figure 5, the two red/underlined intervals are overlapping since they have an address in common. Therefore, SWORD reports a race at the lines of code associated with the program counter stored by the intervals.

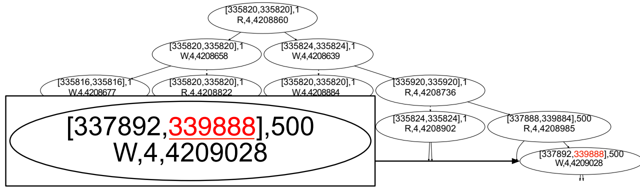
C. Limitations

Although SWORD supports most of the constructs defined by the OpenMP specification, in its current form it cannot analyze programs based on OpenMP tasking. The main limitation for supporting OpenMP tasking is that the current formulation of the offset-span label mechanism does not allow for identifying whether two threads that executed two different tasks are concurrent or not. This is critical to avoid false alarms and missed races. Despite this limitation, programs that employ OpenMP tasking are still rare, thus SWORD can analyze most of the existing OpenMP applications.

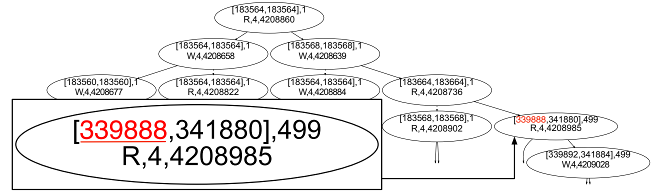
IV. Experimental Results

We evaluate SWORD on two OpenMP microbenchmark suites and four large real-world HPC applications. More specifically, we select DataRaceBench [2] and OmpSCR [25] OpenMP microbenchmarks to show the effectiveness of SWORD in terms of identifying data races. In addition, we use real-world HPC applications to assess its performance and memory overhead. We compare SWORD against the state-of-the-art OpenMP data race checker ARCHER [1].[4] In our

[4] We also performed a preliminary comparison with the latest version of Intel®Inspector XE. We obtained results that are very similar to its comparison with ARCHER from our previous work [1]. Hence, we omit a detailed comparison with Intel®Inspector XE from this paper.



(a) Interval tree for Thread 0



(b) Interval tree for Thread 1

Figure 5: Example interval trees. The red/underlined nodes are the two overlapping intervals that identify the race. The node's fields represent respectively [begin,end] of the interval, count, type of operation, access size, and program counter.

experiments, we run two configurations of ARCHER: with default settings and with the "flush shadow" option enabled. The purpose of enabling this option, which flushes memory between independent parallel regions, is to try to reduce the memory overhead of ARCHER and to have a more fair comparison with SWORD. We also use the default setup of 4 shadow cells per 'line' (see Section II).

We perform our evaluation on a machine with two 12-core Intel Xeon E5-2695v2 processors, 32GB of RAM, and 800GB of SSD storage. The machine runs the TOSS Linux distribution (kernel version 3.10), which is a customized distribution specifically optimized for HPC clusters. We average the measured runtimes and memory overhead of all benchmarks across 10 executions, and we vary the number of threads from 8 to 24. In the experimental results, "baseline" denotes the original benchmark characteristics with data race checking disabled, while "archer" and "archer-low" denote ARCHER in its default and low memory overhead configuration respectively, and "sword" denotes our SWORD tool.

A. DataRaceBench Microbenchmarks

The DataRaceBench microbenchmark suite [2] consists of small OpenMP codes with and without data races; each 'racy' benchmark contains one known data race documented by the authors. We run every tool on all benchmarks and inspect the outcomes; none of the tools report false alarms, and they also successfully identified almost all races. All tools missed the races in benchmarks `indirectaccess{1-4}-orig-yes`. These data races do not manifest along all program paths, and given that both SWORD and ARCHER are dynamic analysis tools that analyze only the executed control flow, they can miss such races. In benchmarks `nowait-orig-yes` and `privatemissing-orig-yes`, SWORD analysis is more complete and it reports races that ARCHER misses for the reasons discussed in Section II. These are all *read-write* data races happening in the same shared variable and parallel region. Because of multiple reads by the same thread, the shadow cells maintained by ARCHER are eventually overwritten, and this information loss causes these races to be missed. SWORD does not suffer from such information loss, and it correctly identifies them. Note that all tools report an additional unknown race in `plusplus-orig-yes`, and SWORD reports an additional unknown race in `privatemissing-orig-yes` as well.

Benchmark	*# of Reported Data Races*		
	archer	**archer-low**	**sword**
c_loopA.badSolution	1	1	1
c_loopB.badSolution1	1	1	1
c_loopB.badSolution2	1	1	1
c_md	1	1	2
c_testPath	2	2	6
cpp_qsomp1	1	1	2
cpp_qsomp2	1	1	2
cpp_qsomp5	1	1	3
cpp_qsomp6	1	1	2

TABLE II: Data races reported in OmpSCR suite.

These are not false alarms, but rather real races that the authors of the benchmarks have failed to document (we have reported this, and anticipate a fix in their next release). Finally, since DataRaceBench benchmarks are small, the runtime and memory overheads are similar among the tools.

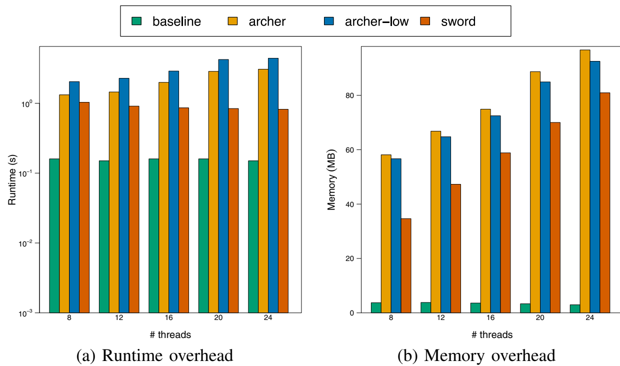
B. OmpSCR Microbenchmarks

The OmpSCR benchmark suite contains known data races that have been documented in previous works [25], [1]. Table II gives the number of data races detected by each tool. (We again omit race-free benchmarks since we verified that none of the tools report false alarms.) SWORD not only identifies the same races as ARCHER, but also detects new undocumented races in the following benchmarks: `c_md`, `c_testPath`, `cpp_qsomp1`, `cpp_qsomp2`, `cpp_qsomp5`, and `cpp_qsomp6`. Our manual inspection confirmed that all these races are real. ARCHER missed these races for all the reasons summarized in Section I.

Figure 6 gives the geometric mean of the runtime and memory overheads to indicate the overall tendency of the values, considering the large gaps in execution time and memory usage among the different benchmarks. The runtime overhead is small for all tools, while the relative memory overhead is large due to small baseline, but still less than 100 MB for all tools. Also note that the memory overhead of SWORD is constantly around 3.3 MB per thread, as we indicated in Section III. When compared, the runtime and memory overhead of the SWORD data collection is lower than ARCHER in both configurations. The plots do not include the runtime and memory overhead of the offline data race detection algorithm, which may increase the total amount of resources needed by SWORD for a complete analysis.

Benchmark	baseline(s)	archer(s)	archer-low(s)	sword				
				DA(s)	OA(s)	MT(s)	#PR	LS
c_fft	0.13	0.81	0.84	0.52	2.09	1.34	2	2.4MB
c_fft6	0.03	0.14	0.15	0.12	0.12	0.12	1	122kB
c_jacobi01	0.9	19.83	20.91	2.57	2.06	1.33	2	51MB
c_jacobi02	0.89	19.64	20.38	2.59	0.63	0.63	1	51MB
c_loopA.badSolution	0.03	0.47	1.59	0.18	3.16	0.35	100	394kB
c_loopA.solution1	0.03	0.65	2.76	0.36	5.88	0.22	200	981kB
c_loopA.solution2	0.03	0.3	0.39	0.27	0.14	0.14	1	452kB
c_loopA.solution3	0.03	0.3	1.43	0.23	2.33	0.17	100	458kB
c_loopB.badSolution1	0.03	0.47	1.62	0.3	3.03	0.14	100	398kB
c_loopB.badSolution2	1.79	4.08	5.26	2.26	3.09	0.15	100	390kB
c_loopB.pipelineSolution	0.03	0.28	0.32	0.25	0.14	0.14	1	462kB
c_lu	0.04	10.5	15.81	0.83	25.35	0.28	499	20MB
c_mandel	0.08	5.06	5.05	0.37	0.1	0.1	1	81kB
c_md	0.47	80.87	84.47	3.65	0.55	0.17	21	1.5MB
c_pi	0.02	0.14	0.17	0.14	0.11	0.11	1	81kB
c_qsort	0.04	0.23	0.33	0.14	0.27	0.12	10	125kB
c_testPath	0.03	0.26	0.33	0.26	0.09	0.09	1	81kB
cpp_qsomp1	1.38	259.9	264.32	5.46	1.76	1.76	1	321MB
cpp_qsomp2	1.38	262.8	263.19	5.39	1.82	1.82	1	303MB
cpp_qsomp5	14.27	41.54	41.51	55.44	16.47	16.47	1	204MB
cpp_qsomp6	1.52	263.51	263.16	5.36	1.93	1.93	1	316MB
Mean	1.1	46.28	47.33	4.13	–	–	–	–
Median	0.04	0.81	2.76	0.37	–	–	–	–
Geometric Mean	0.15	0.81	2.76	0.37	–	–	–	–

TABLE III: Overheads on the OmpSCR suite executed with 24 threads, including the execution time of the parallel offline analysis. Column **baseline** is the baseline runtime; **archer** is the ARCHER runtime; **archer-low** is the low memory overhead ARCHER configuration runtime; **DA** is the total dynamic analysis runtime including logging; **OA** is the offline analysis runtime when executed on just one node (24 threads); **MT** (Max Time) is the time taken by 24 threads to analyze the region of the offline traces that have the maximum amount of trace info (MT can reduce with more threads); **#PR** is the number of independent parallel regions to analyze; **LS** is the amount of storage required to store the generated log files.



(a) Runtime overhead (b) Memory overhead

Figure 6: Geometric mean of runtime and memory overhead for OmpSCR suite; the number of threads varies from 8 to 24.

Table III shows the overheads of the offline data race checking with SWORD compared to the two ARCHER configurations. The runtime overhead depends on the size of log files and the number of parallel regions the algorithm has to analyze for each benchmark. We distributed the offline analysis across a cluster of nodes, and the offline data race detection in that case typically lasts from a few milliseconds up to a few seconds (column **MT**). Moreover, even running the entire offline analysis on a single node (24 threads) takes less than a minute for all benchmarks (column **OA**). We omit the memory overhead for the dynamic analysis since it is negligible given the small size of the benchmarks. While for most of the benchmarks the dynamic analysis terminates quickly and does not differ much from the runtime overhead of ARCHER, for some the offline analysis takes a considerable amount of time.

C. HPC Benchmarks

We assess the performance and memory overhead of SWORD using four small to large-size HPC benchmark codes. We use three codes, namely AMG2013, LULESH, and miniFE, from the CORAL benchmark suite [26], while the fourth code HPCCG is a part of the Mantevo project [27]. These codes model scientific problems and simulations, and their size ranges from tens to hundreds of thousands of lines of code. We also leverage AMG2013 to evaluate the overheads of the tools with an increasing problem size. AMG2013 is a parallel algebraic multigrid solver for linear systems arising from problems on unstructured grids. Therefore, we perform the evaluation using 4 different grid sizes: 10^3 (AMG2013_10), 20^3 (AMG2013_20), 30^3 (AMG2013_30), and 40^3 (AMG2013_40).

Table IV shows the number of data races detected by each tool. Note that none of the tools report false alarms. Both tools find one race in HPCCG, which happens in a parallel region where all threads are writing the same value into a shared variable. While this race may seem harmless, it in fact results in undefined behavior based on the C/C++ standard,

Benchmark	# of Reported Data Races archer	archer-low	sword
miniFE	0	0	0
HPCCG	1	1	1
LULESH	0	0	0
AMG2013_10	4	4	14
AMG2013_20	4	4	14
AMG2013_30	4	4	14
AMG2013_40	OOM	OOM	14

TABLE IV: Data races reported in HPC benchmarks. OOM indicates that a tool ran out of memory during the analysis.

and compiler optimizations could unpredictably modify the outcome of this program [1], [28]. ARCHER detects 4 known races in smaller-scale AMG2013 runs [1], while it runs out of memory at large scale. SWORD both completes the analysis at large scale and detects 10 additional races missed by ARCHER. These races happen in the same large parallel region (around 400 LOC) as the others, and they are all the same type of read-write races. As before, ARCHER misses them since it maintains only a limited number of previous accesses, while SWORD detects them since it logs every memory access.

Figure 7 shows the slowdown and memory overhead of the tools on the HPC benchmarks. ARCHER in both configurations exhibits a larger slowdown than SWORD as we are increasing the number of threads. The "archer-low" configuration flushes the shadow memory in-between independent parallel regions, and the plots show that this slightly reduces the memory overhead, but it also increases the runtime overhead because of the additional operations to release memory pages. SWORD, on the other hand, exhibits better scaling, typically resulting in a faster dynamic analysis than ARCHER, with the exception of LULESH (see Figure 7c). LULESH executes a large number of parallel regions and barriers that significantly increase the number of I/O operations during the log collection phase of SWORD. The plots show that the memory overhead of ARCHER depends on the baseline memory consumption and is around 5–7× of the baseline. On the other hand, SWORD's memory overhead is bounded since it depends only on the number of threads (it is around 3.3 MB per thread) and not the baseline. Figure 8 further analyzes this behavior by varying the problem input size of AMG2013. This clearly illustrates a major advantage of SWORD: as the baseline memory consumption increases ARCHER runs out of memory, while SWORD's bounded memory overhead allows it to finish its analysis successfully.

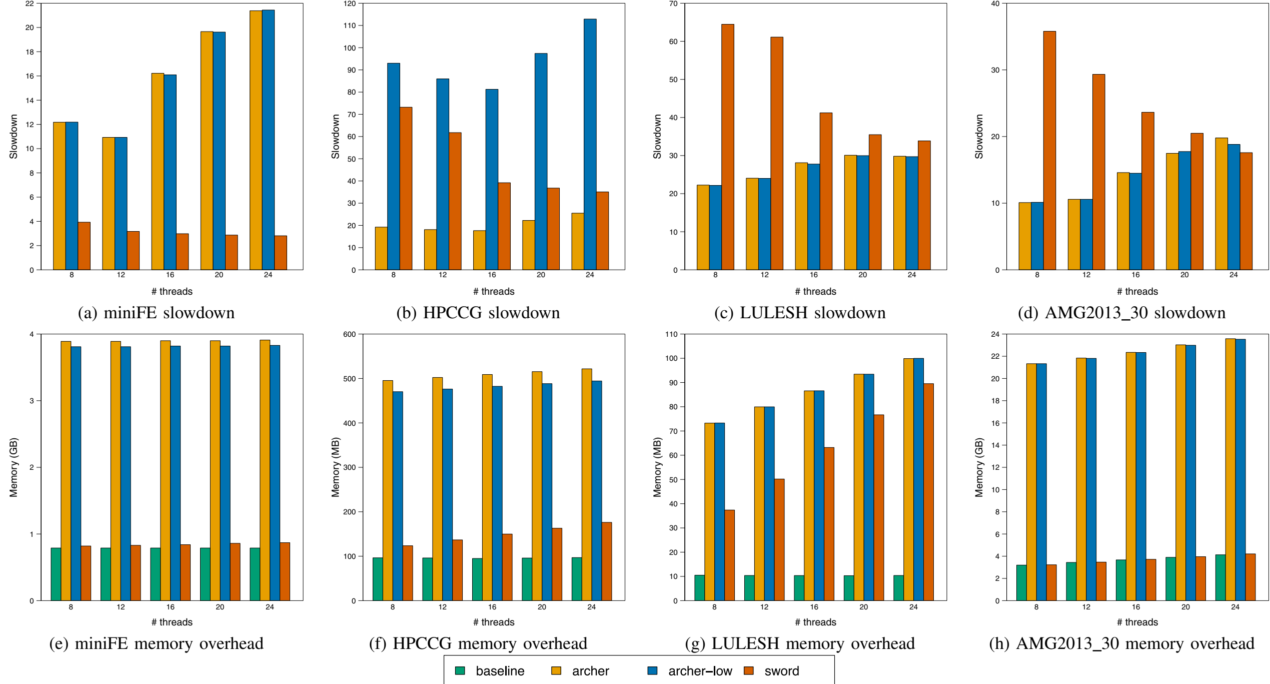
As Figure 7 and Figure 8 indicate, SWORD's dynamic analysis (log collection) is typically faster than ARCHER at larger scales. However, we need to take the offline analysis execution time into account to represent the total runtime overhead of SWORD. Table V shows the overheads of the tools including the offline analysis of SWORD. The overall analysis runtime of SWORD for HPCCG, including the offline data race detection process, is less than 2 minutes if executed on a single node and can be reduced to several seconds if executed on a cluster; the latter is not significantly different from ARCHER. On the other hand, SWORD is about 4 times faster than ARCHER on miniFE. On LULESH, the SWORD's dynamic analysis is slower compared to ARCHER since LULESH generates almost 300,000 independent parallel regions which increase the I/O operations, thereby slowing down the data collection phase. Subsequently, the SWORD's offline analysis takes more than 24 hours, because of the large number of regions to analyze. For our experiments we used 24 cores per node, each core generating the interval-tree of a different thread. While the tree generation cannot be efficiently parallelized since it would require the use of locks, we could significantly reduce this large offline analysis time by using many more cores for the comparison of the interval trees of different threads. The most interesting case is AMG, where ARCHER runs out of memory at large problem sizes and does not complete its analysis, while SWORD is able to collect all the data at runtime and perform the offline data race detection process. Even though the SWORD's offline analysis takes about an hour when executed on a single node, it does not take more than a few minutes when executed on a cluster, and the data race detection is more complete than ARCHER.

V. RELATED WORK

Data race detection is a widely studied problem in concurrent programming. Netzer and Miller provide a good survey of general approaches for data race detection [29]. A number of different techniques have been proposed, including static analysis [3], [5], [6], [30], [31], dynamic analysis [9], [12], [32], and hybrid analysis [33]. These are not directly applicable to OpenMP, as they fail to consider the runtimes and internal actions of OpenMP programs. A complete survey of data race detection methods is beyond the scope of this work; in this section we focus on works that either address OpenMP race checking, or are more closely related.

There has been prior work on OpenMP race checking, including the use of dynamic analysis (e.g., [34]) and symbolic analysis (e.g., [35]). Our prior work [1] and its precursor [36] document the success of ARCHER in practical OpenMP race checking, and this observation is also in line with that in a recent study [2]. The main weakness of ARCHER is its memory consumption, which can be 6× the amount of memory needed by the innate (unmodified) application. ARCHER does provide an option to release some of the allocated memory in between independent parallel regions, thereby often reducing the memory overhead by around 30%. However, as we show in Section IV, even this memory reduction is insufficient for dealing with large OpenMP applications that allocate up to 90% of the available memory in each compute node.

There have been many efforts that make race checking efficient by exploiting structured parallelism found in languages such as Cilk [37], X10 [38], or Habanero Java [39]. These techniques are not directly applicable to OpenMP. Similarly to SWORD, Wilcox et. al. [40] propose an approach to reduce memory overhead by employing array summarization, where array accesses can be summarized into the same shadow-cell. This approach reduces the memory overhead by about 30% for



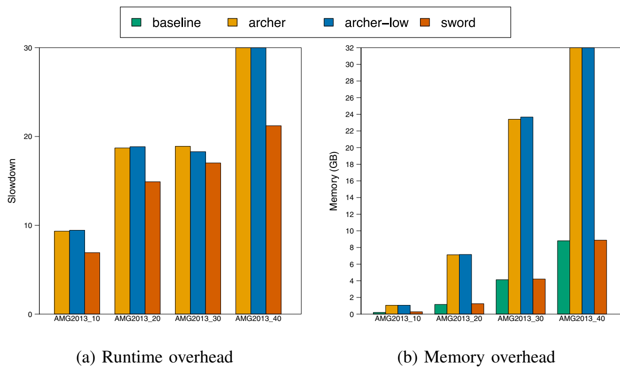
(a) miniFE slowdown (b) HPCCG slowdown (c) LULESH slowdown (d) AMG2013_30 slowdown

(e) miniFE memory overhead (f) HPCCG memory overhead (g) LULESH memory overhead (h) AMG2013_30 memory overhead

Figure 7: Relative slowdown and memory overhead compared to the baseline for HPC benchmarks.

Benchmark	baseline(s)	archer(s)	archer-low(s)	sword				
				DA(s)	OA(s)	MT(s)	#PR	LS(GB)
miniFE	4.7	101.4	101.6	13.3	8.1	4.3	28	1.1
HPCCG	0.4	10.5	46.3	14.4	84.9	2.3	898	2.8
LULESH	3.9	116.1	115.6	131.7	>24h	40.0	300,000	9.8
AMG2013_10	2.2	19.8	20.1	14.9	811.0	5.4	1,272	2.4
AMG2013_20	7.7	149.1	147.2	115.9	2,116.0	41.0	1,527	20.0
AMG2013_30	23.8	471.4	448.2	418.7	3,153.0	133.2	1,575	57.0
AMG2013_40	57.2	OOM	OOM	1,251.4	3,871.0	180.2	2,036	162.0

TABLE V: Overheads on the HPC benchmarks executed with 24 threads, including the execution time of the parallel offline analysis. See Table III for the explanation of columns. OOM indicates that the tool ran out of memory during the analysis.



(a) Runtime overhead (b) Memory overhead

Figure 8: Runtime and memory overhead on AMG2013 with varying problem size executed with 24 threads.

array-intensive applications. However, it does not overcome the happens-before and shadow-memory limitations explained in Section II.

VI. Conclusions

Given the growing importance of OpenMP for harnessing on-node parallelism, data races in production-scale OpenMP programs present a looming threat to reliable parallel software design. Today's happens-before-relation-based race checkers for OpenMP (notably Archer, the best in its class) are highly memory inefficient, needing at least five times (and six times in practice) more memory than the application itself. Despite such a large memory overhead, they also miss a significant number of data races due to either schedule-based race masking or shadow-cell eviction.

In contrast, in our new work embodied in the tool Sword, the online analysis can be carried out using a memory buffer of under 3 megabytes in size. Traces collected in this buffer are compressed, and written out to log files, where the offline analysis based on stepping an operational semantics model takes over. This algorithm is also memory efficient, being based on novel streaming algorithms and state-of-the-art interval tree data structures to merge traces and check for

races. ILP-based constraint-solving further reduces the overhead of detecting overlapping accesses. Overall, SWORD is at least 1,000 times more memory-efficient than ARCHER, thus virtually guaranteeing the absence of out-of-memory errors. For instance, we could not finish checking the AMG2013 benchmark at large scale using ARCHER, while with SWORD this was easily accomplished.

We present extensive experimental results that demonstrate these features of SWORD as well as its overall superior performance as well as race coverage. We performed the experiments on a recently published OpenMP benchmark suite [2] as well as all previous data race checking benchmarks on which ARCHER was run. Experimental results demonstrate that SWORD is comparable to ARCHER even on examples where the memory pressure is not an issue. SWORD is also sound and complete with respect to data race checking in the absence of data-dependent control flow variations. Last but not least, SWORD has actually found races missed by ARCHER as well as some of the feasible races that are not documented in a recent study [2].

While SWORD's dynamic analysis is overall faster than ARCHER, its offline data race analysis can sometimes take a long time, especially at very large scales. This slow-down can be mitigated through the development of novel parallel algorithms, which we relegate to future work. We also plan to extend SWORD's approach to target regions that are offloaded on accelerators, as well as accommodate tasking.

In conclusion, SWORD is currently the tool of choice for checking data races in large-scale OpenMP programs. In production use, a user of SWORD may employ available techniques to systematically explore the execution-space of their application, and attempt to check for data races within these executions. They can carry this out without worrying about out-of-memory errors—even when checking their applications on production-level inputs. In the process, they will also obtain superior race coverage than any available OpenMP race checker.

REFERENCES

[1] S. Atzeni, G. Gopalakrishnan, Z. Rakamarić, 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 *IPDPS*, 2016.

[2] 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 *Supercomputing*, 2017, pp. 11:1–11:14.

[3] P. Chatarasi, J. Shirako, M. Kong, and V. Sarkar, "An extended polyhedral model for SPMD programs and its use in static data race detection," in *Languages and Compilers for Parallel Computing (LCPC)*, 2016.

[4] P. Pratikakis, J. S. Foster, and M. Hicks, "LOCKSMITH: Practical static race detection for C," *TOPLAS*, vol. 33, no. 1, pp. 3:1–3:55, Jan. 2011.

[5] J. W. Voung, R. Jhala, and S. Lerner, "RELAY: Static race detection on millions of lines of code," in *ESEC/FSE*, 2007, pp. 205–214.

[6] D. Engler and K. Ashcraft, "RacerX: Effective, static detection of race conditions and deadlocks," in *SOSP*, 2003, pp. 237–252.

[7] A. Jannesari, K. Bao, V. Pankratius, and W. F. Tichy, "Helgrind+: An efficient dynamic race detector," in *IPDPS*, 2009, pp. 1–13.

[8] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: Data race detection in practice," in *Workshop on Binary Instrumentation and Applications (WBIA)*, 2009, pp. 62–71.

[9] K. Serebryany and D. Vyukov, "ThreadSanitizer, a data race detector for C/C++ and Go," https://github.com/google/sanitizers.

[10] U. Banerjee, B. Bliss, Z. Ma, and P. Petersen, "Unraveling data race detection in the Intel Thread Checker," in *STMCS*, 2006.

[11] J. Protze, J. Hahnfeld, D. H. Ahn, M. Schulz, and M. S. Müller, "OpenMP tools interface: Synchronization information for data race detection," in *International Workshop on OpenMP (IWOMP)*, 2017, pp. 249–265.

[12] C. Flanagan and S. N. Freund, "FastTrack: Efficient and precise dynamic race detection," in *PLDI*, 2009, pp. 121–133.

[13] J. Huang, C. Zhang, and J. Dolby, "CLAP: Recording local executions to reproduce concurrency failures," in *PLDI*, 2013, pp. 141–152.

[14] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan, "Sound predictive race detection in polynomial time," in *POPL*, 2012.

[15] S. Atzeni and G. Gopalakrishnan, "An Operational Semantic Basis for Building an OpenMP Data Race Checker," in *23rd International Workshop on High-Level Parallel Programming Models and Supportive Environments*, 2018, IEEE Xplore Digital Library.

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

[17] 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 *International Workshop on OpenMP (IWOMP)*, 2013, pp. 171–185.

[18] F. Mattern, "Virtual time and global states of distributed systems," in *Parallel and Distributed Algorithms Conference*, 1988, pp. 215–226.

[19] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004, pp. 75–86.

[20] M. F. Oberhumer, "LZO," http://www.oberhumer.com/lzo, 2012.

[21] Google, "Snappy," https://google.github.io/snappy, 2011.

[22] Y. Collet, "LZ4," https://lz4.github.io/lz4, 2011.

[23] J. Gama, *Knowledge Discovery from Data Streams*. Chapman & Hall/CRC, 2010.

[24] L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees," in *Annual Symposium on Foundations of Computer Science (SFCS)*, 1978, pp. 8–21.

[25] A. J. Dorta, C. Rodriguez, and F. d. Sande, "The OpenMP source code repository," in *EMPDP*, 2005, pp. 244–250.

[26] "CORAL benchmark codes," https://asc.llnl.gov/CORAL-benchmarks.

[27] "Mantevo," https://mantevo.org, 2013.

[28] H.-J. Boehm, "How to miscompile programs with "benign" data races," in *HotPar*, 2011, pp. 3–3.

[29] R. H. B. Netzer and B. P. Miller, "What are race conditions? Some issues and formalizations," *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 1, pp. 74–88, Mar. 1992.

[30] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta, "Fast and accurate static data-race detection for concurrent programs," in *Conference on Computer Aided Verification (CAV)*, 2007, pp. 226–239.

[31] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for Java," in *PLDI*, 2006, pp. 308–319.

[32] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multi-threaded programs," in *Symposium on Operating Systems Principles (SOSP)*, 1997, pp. 27–37.

[33] R. O'Callahan and J.-D. Choi, "Hybrid dynamic data race detection," in *PPoPP*, 2003, pp. 167–178.

[34] O.-K. Ha, I.-B. Kuh, G. M. Tchamgoue, and Y.-K. Jun, "On-the-fly detection of data races in OpenMP programs," in *PADTAD*, 2012.

[35] H. Ma, S. R. Diersen, L. Wang, C. Liao, D. Quinlan, and Z. Yang, "Symbolic analysis of concurrency errors in OpenMP programs," in *International Conference on Parallel Processing (ICPP)*, 2013.

[36] J. Protze, S. Atzeni, D. H. Ahn, M. Schulz, G. Gopalakrishnan, M. S. Müller, I. Laguna, Z. Rakamarić, and G. L. Lee, "Towards providing low-overhead data race detection for large OpenMP applications," in *LLVM Compiler Infrastructure in HPC*, 2014, pp. 40–47.

[37] G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark, "Detecting data races in Cilk programs that use locks," in *Symposium on Parallel Algorithms and Architectures (SPAA)*, 1998, pp. 298–309.

[38] T. Yuki, P. Feautrier, S. V. Rajopadhye, and V. Saraswat, "Checking race freedom of clocked X10 programs," *CoRR*, vol. abs/1311.4305, 2013.

[39] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav, "Efficient data race detection for async-finish parallelism," in *RV*, 2010, pp. 368–383.

[40] J. R. Wilcox, P. Finch, C. Flanagan, and S. N. Freund, "Array shadow state compression for precise dynamic race detection," in *ASE*, 2015.