LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-750746

Runtime and Memory Evaluation of Data Race Detection Tools

P. Lin, C. Liao, M. Schordan, I. Karlin

May 3, 2018

ISOLA 2018
LIMASSOL, Cyprus
November 5, 2018 through November 9, 2018

Disclaimer

Runtime and Memory Evaluation of Data Race Detection Tools *

Pei-Hung Lin, Chunhua Liao, Markus Schordan, Ian Karlin

Lawrence Livermore National Laboratory, Livermore CA 94550, USA

Abstract. An analysis tool's usefulness depends on whether its runtime and memory consumption remain within reasonable bounds for a given program. In this paper we present an evaluation of the memory consumption and runtime of four data race detection tools: Archer, ThreadSanitizer, Helgrind, and Intel Inspector, using DataRaceBench version 1.1.1 using 79 microbenchmarks. Our evaluation consists of four different analyses: (1) runtime and memory consumption of the four data race detection tools using all DataRaceBench microbenchmarks, (2) comparison of the analysis techniques implemented in the evaluated tools, (3) for selected benchmarks an in-depth analysis of runtime behavior with CPU profiler and the identified differences, (4) data analysis to investigate correlations within collected data. We also show the effectiveness of the tools using three quantitative metrics: precision, recall, and accuracy.

Keywords: data race detection, DataRaceBench, evaluation, benchmark

1 Introduction

The widespread use of threaded programming models and the increasing on-node parallelism necessitate effective and efficient tools to detect and fix data races within limited time and system resources. A data race occurs when two or more threads perform simultaneous conflicting data accesses to the same memory location without proper synchronization and at least one access is a write. Data race bugs may lead to unpredictable results of a parallel program even with the exact same input. Due to this behavior, the difficulties in detecting and fixing data race bugs can greatly reduce programming productivity. Therefore there is an increasing demand for data race detection tools and many industrial and research efforts are providing tools with data race detection capabilities. An effective data race detection tool needs to have high accuracy in detecting and reporting the data race instances. In the ideal case, the tool is efficient in exploiting shared system resources and detects data races in the short time.

The DataRaceBench benchmark suite [12] was specifically designed for systematic evaluation of data race detection tools with focus on the OpenMP parallel programming model. The development team of DataRaceBench has presented initial evaluation results using quantitative metrics for four selected data race detection tools: Archer, ThreadSanitizer, Helgrind and Intel Inspector.

The evaluation with DataRaceBench lead to the following discoveries:

* Prepared by LLNL under Contract DE-AC52-07NA27344. LDRD 17-ERD-023. LLNL-CONF-750746

- OpenMP awareness is a necessity for detecting data races in programs using the OpenMP programming model. Even if an OpenMP runtime library is implemented using Pthreads, a tool that only considers pthread semantics will miss certain details of the OpenMP semantics, leading to false alarms (false positives) in the analysis.
- There is no existing dynamic tool supporting data race detection for SIMD directives.
- With all four of the evaluated dynamic data race detection tools, it is necessary to run the tool multiple times (with the exact same input) because some data races only occur with certain thread schedules. Note that this adds to the overall runtime of dynamic analysis tools in practice.
- Dynamic testing tools can be sensitive to the number of threads used in testing. DataRaceBench includes microbenchmarks that are specifically designed to only have data races with a certain number of threads.

DataRaceBench is useful in evaluating the effectiveness of data race detection tools, but it remains difficult to predict how fast a tool can detect a race and how many system resources it requires.

This paper continues the experiments from [12] to include more details about the runtime and memory consumption of the selected data race detection tools. We use these metrics as basis to determine the most efficient race detection tools.

We present results for four kinds of evaluations. First, we analyze the differences among microbenchmarks in DataRaceBench and show characteristics in microbenchmarks. Second, we compare the selected tools to understand their different runtime behaviors. Third, we use additional evaluation tools to investigate the differences between two selected data race detection tools: ThreadSanitizer and Archer. For Intel Inspector we also show the differences between the default configuration and the max-resource configuration. Finally, we apply a data analysis tool to investigate correlations within the collected experimental results.

The following contributions are be presented in this paper:

- We present an updated evaluation of the tools accuracy extended with memory consumption results for DataRaceBench version 1.1.1 with additional microbenchmarks.
- We collect the runtime overheads of the selected tools when run on DataRaceBench microbenchmarks.
- We show three different evaluations with analysis using DataRaceBench. We provide a tool comparison using quatifiable metrics, detailed analysis to understand the behaviors of specific tools, and analysis of our collected data with a data analysis tool to show what features of the benchmarks impact our metrics the most.

The rest of the paper is organized as follows: in Section 2 we present the DataRaceBench suite. In Section 3 we describe the selected data race detection tools and the analysis techniques they use. In Section 4 we present the experimental results and analyses. In Section 5 we discuss the related work and in section 6 we conclude.

2 DataRaceBench Suite

DataRaceBench was designed to capture possible data race patterns in OpenMP programs and serve as a measurement suite to evaluate the accuracy and overhead of data

race detection tools. The development of the DataRaceBench suite followed specific design guidelines to achieve its design goals:

- Each microbenchmark should be compact and representative.
- Each program should be self-contained so they can be easily used as part of regression tests for developing data race detection tools.
- Each microbenchmark has a main function to support dynamic data race detection.
- The benchmark suite contains microbenchmarks that allow arbitrary problem sizes to probe static data race detection tools.
- The benchmark suite categorizes code patterns (or properties) of data races in OpenMP programs in its microbenchmark collection.
- The microbenchmarks are divided into two categories: microbenchmarks with known data races and those which are known to be data race free.
- If possible, a program in the category with known data races (race-yes set) should only contain a single pair of source locations that cause data races.
- All microbenchmark programs have built-in input data. A subset of microbenchmarks comes with one additional variant using C99 variable-length arrays to allow probing the impact of changing input data sizes.

The collected microbenchmarks we selected from four major sources: (1) regression test cases from the auto-parallelization tool AutoPar [13]; (2) parallel optimization variants, generated by polyhedral optimizations; (3) data access patterns found in development branches of real scientific applications; and (4) microbenchmarks built by the DataRaceBench developers.

In order to represent different patterns found in OpenMP programs with and without data races, a list of property labels are assigned to the collected microbenchmarks in the DataRaceBench, as shown in Table 1.

Property labels for race-yes set	**Property labels for race-no set**
Y1: Unresolvable dependencies	N1: Embarrassingly parallel
Y2: Missing data sharing clauses	N2: Use of data sharing clauses
Y3: Missing synchronization	N3: Use of synchronization
Y4: SIMD data races	N4: Use of SIMD directives
Y5: Accelerator data races	N5: Use of accelerator directives
Y6: Undefined behaviors	N6: Use of special language features
Y7: Numerical kernel data races	N7: Numerical kernels

Table 1: Property labels of microbenchmarks

The DataRaceBench suite is available at `https://github.com/LLNL/dataracebench`. The version 1.1.1 used in this paper, uses 79 microbenchmarks to represent all the property labels shown in Table 1.

3 Data Race Detection Tools

In this paper, we use the same race detection tools used in [12]: Helgrind,ThreadSanitizer, Archer and Intel Inspector. All these tools use dynamic analysis to find data races during the runtime. Helgrind and ThreadSanitizer are recommended for applications using

the POSIX thread API. The versions of the selected data race detection tools we used are listed in Table 2, with the respective compilers used (either to build the tools, compile the microbenchmarks, or both).

Tool	**Version**	**Compiler**
Helgrind	3.12.0	GCC 4.9.3
ThreadSanitizer	4.0.1	Clang/LLVM 4.0.1
Archer	towards_tr4 branch	Clang/LLVM 4.0.1
Intel Inspector	2017 (build 475470)	Intel Compiler 17.0.2

Table 2: data race detection tools: versions and compilers

Helgrind. Helgrind [1] is a Valgrind-based error detection tool for C, C++ and Fortran programs with POSIX threads. It targets three classes of errors: a) Misuses of the Pthreads API: Helgrind intercepts POSIX thread function calls to detect errors and provides stack trace information for the detected error. b) Potential deadlock from lock ordering problems: The order in which threads acquire locks is monitored by Helgrind to detect potential deadlocks. c) Data races: Helgrind follows the "happen-before" tracking and intercepts a selected list of events. It monitors all memory access and builds a directed acyclic graph that represents the collective happens-before dependencies. Analysis with Helgrind can have slowdowns on the order of 100:1, as noted in the documentation. In [15] a slowdown of 20× to 30× is reported. We use Valgrind version 3.12.0 (built with GCC 4.9.3) and GCC version 4.9.3 to compile the microbenchmarks.

ThreadSanitizer. ThreadSanitizer (Tsan) [2] is a runtime data race detector developed by Google. ThreadSanitizer is now part of the LLVM and GCC compilers to enable data race detection for C++ and Go code. Every memory access is instrumented by ThreadSanitizer and every aligned 8-byte word of application memory is mapped to N shadow words through direct address mapping (N is configurable to 2, 4 and 8). A state machine that updates the shadow state at every memory address iterates over all stored shadow words. A warning message is printed when one shadow word constitutes a data race with the other shadow word. The memory overhead of ThreadSanitizer comes from four main sources [23]: (1) a constant size buffer that stores segments (a segment has a sequence of events of one thread that contains only memory access events), including stack traces, (2) vector time clocks attached to each segment, (3) Per-ID state, and (4) segment sets and locksets. The segment sets and locksets may potentially consume arbitrary large amount of memory. On an average Google unit test the memory overhead is within a factor of 3 to 4 (compared to a native run).

Archer. Archer is an OpenMP data race detector that exploits ThreadSanitizer to achieve scalable happen-before tracking. It takes the LLVM-based tooling approach to

[1] `http://valgrind.org/docs/manual/hg-manual.html`
[2] `https://github.com/google/sanitizers`

develop LLVM passes within the LLVM package. In addition to the dynamic analysis performed by ThreadSanitizer, Archer adopts static analysis to categorize OpenMP regions into *guaranteed race-free* and *potentially racy*. LLVM passes are designed to identify guaranteed sequential regions within OpenMP code. Memory accesses within OpenMP parallelizable loops, detected by LLVM's Polly [3], are black-listed for the dynamic analysis checking. The unmodified ThreadSanitizer reports a high number of false positives in OpenMP code caused by potential confusion in OpenMP runtime actions. To avoid the confusion for better OpenMP race detection, Archer includes a customized ThreadSanitizer and employs ThreadSanitizer's annotation API to identify the synchronization points within OpenMP runtime.

In this paper, we use the development branch of Archer built based on LLVM version 4.0.1[4]. The OpenMP runtime support for Archer is from the OMPT[5].

Intel Inspector. Intel Inspector [6] is a dynamic analysis tool that detects threading and memory errors in C, C++ and Fortran codes. It supersedes Intel's Thread Checker tool [21,18], with added memory error checking. Supported thread errors include race conditions and deadlocks.

We used a commercial version of Intel Inspector. This tool provides three different levels of analysis. The widest scope maximizes the load on the system for more thorough analysis but has higher analysis overhead. Intel Inspector also allows customized configuration in the data race analysis, e.g. byte granularity, stack frame depth and resources used. By default, Intel uses four bytes for monitoring memory accesses, a stack frame depth of 1, without exploiting maximum resources. Setting the access granularity to be a single byte and increasing the stack frame depth to 16 leads to exploiting the maximum resources and increases precision. One limitation is that this version does not support GCC's OpenMP runtime and may report false positives for OpenMP codes compiled by GCC. Therefore, we used Intel C/C++ compilers (with the supported Intel OpenMP runtime) to compile our microbenchmark programs. We also turned off optimizations to keep the best possible debugging information available for dynamic analysis. The graphic user interface of Intel Inspector can provide analysis time overhead and memory overhead to help quickly estimate the time and memory required for the data race analysis. The Livermore computing center reports that slowdowns in the order of 2x to 160x from Intel Inspector can be expected. It is suggested that users choose a small, representative workload, when running Intel Inspector rather than a full production workload.

4 Evaluation

The evaluation in this paper is based on DataRaceBench, version 1.1.1, with seven new microbenchmarks compared to version 1.0.1 used in [12]. Several microbenchmarks in DataRaceBench are designed to have data races only when running with a specific number of OpenMP threads. To allow tools to detect such data races, the experiments

[3] `https://polly.llvm.org`

[4] `https://github.com/PRUNERS/archer` 5ad2f47bc8ca8aad006a82a567179d2e0ce1ba75

[5] `https://github.com/OpenMPToolsInterface/LLVM-openmp.git` 6e7140bf94d178f719200a6543558d7ae079183b

[6] `https://software.intel.com/en-us/intel-inspector-xe`

were run with different number of OpenMP threads from a selected list of numbers: (3,36,45,72,90,180,256). Different array sizes (32,64,128,256,512,1024) are provided on the command line to allocate arrays of different sizes in the microbenchmarks (var-length set). There are 79 microbenchmarks in total and 16 of them are in the var-length set, a set of benchmarks that allow users to define the length of arrays used in the benchmark as parameter on the command line.

Each tool is run five times for every microbenchmark with the exact same input and independent of any other run. The reason why we run the tools several times is that the analysis results can differ dependent on the thread schedule. A total of $(5\times7\times6\times16)+(5\times7\times1\times63) = 5565$ tests were performed by each tool. The evaluation collects the detected number of data races, testing time, and memory consumption, and computes the quantitative metrics for all microbenchmarks using the selected data race detection tools.

Our testing platform is the Quartz cluster hosted at the Livermore Computing Center. Each computation node of the cluster has two Intel 18-core Xeon E5-2695 v4 processors with hyper threading support (18 cores $\times$ 2 sockets $\times2$ = 72 threads in total). The details of each data race detection tool and the used compilers are shown in Table 2.

The test script provided as part of the DataRaceBench distribution, collects the execution time as wall time (millisecond precision). The memory is not limited and we measure the maximum amount of memory that is used by the tool when analyzing the execution of the microbenchmark. None of the benchmarks fork processes. The computation nodes are configured to use no swap space. We use the Linux command `/usr/bin/time -f "%M"` to record the maximum resident set size (RSS), the portion of memory occupied by a process that is held in main memory, of the data race detection process during its lifetime. All the collected information is recorded in csv file format for further post processing and analysis.

4.1 Data Race Detection Report

Table 3 reports the data race detection results for the seven new microbenchmarks in DataRaceBench version 1.1.1 following the same format as in [12]. The results include the range of minimum and maximum number of races (min race - max race) and the type of result in true positive (TP), false negative (FN), true negative (TN) or false positive (FP).

ID	Microbenchmark Program	R	Data Race Detection Tools							
			Helgrind		ThreadSanitizer		Archer		Intel Inspector	
			min race	max race type	min race	max race type	min race	max race type	min race	max race type
73	doall2-orig-yes.c	Y	13	18 TP	3	120 TP	2	126 TP	1	1 TP
74	flush-orig-yes.c	Y	9	13 TP	9	9 TP	1	3 TP	1	1 TP
75	getthreadnum-orig-yes.c	Y	6	13 TP	3	124 TP	2	255 TP	0	1 TP/FN
76	flush-orig-no.c	N	10	11 FP	5	8 FP	0	0 TN	0	0 TN
77	single-orig-no.c	N	6	12 FP	2	11 FP	0	0 TN	0	0 TN
78	taskdep2-orig-no.c	N	26	123 FP	2	11 FP	0	0 TN	0	0 TN
79	taskdep3-orig-no.c	N	21	134 FP	1	14 FP	0	0 TN	0	0 TN

Table 3: Data race detection report for the seven new microbenchmarks. See [12] for the microbenchmarks 1-72. Column R shows whether a program contains a data race.

Table 4 summarizes the numbers of true/false positive and true/false negative results for all 79 microbenchmarks. In Table 5 we show the usual evaluation metrics for precision, recall and accuracy for the selected data race detection tools. The results show that the OpenMP-aware tools Archer and Intel Inspector have higher accuracy than Helgrind and ThreadSanitizer. Archer has the smallest range in the accuracy. Intel Inspector with max resources configuration has both higher value and smaller range in recall and accuracy, but a slightly larger range in precision compared to the default configuration.

Tool	Race:Yes			Race:No		
	TP	TP/FN	FN	TN	TN/FP	FP
Helgrind	41	0	2	1	0	35
ThreadSanitizer	41	0	2	1	0	35
Archer	36	5	2	33	3	0
Intel Inspector default	12	24	7	35	1	0
Intel Inspector max resources	32	9	2	33	3	0

Table 4: Positive and negative results of the tools

Tool	Precision		Recall		Accuracy	
	min	max	min	max	min	max
Helgrind	0.539	0.539	0.953	0.953	0.532	0.532
ThreadSanitizer	0.539	0.539	0.953	0.953	0.532	0.532
Archer	0.923	1.000	0.837	0.953	0.873	0.975
Intel Inspector default	0.923	1.000	0.279	0.837	0.595	0.911
Intel Inspector max resource	0.914	1.000	0.744	0.953	0.823	0.975

Table 5: Metrics for the tools

4.2 Runtime Behavior Analysis

For analyzing the runtime behavior of the analysis tools, we determine a baseline by collecting measurements from running the DataRaceBench microbenchmarks using GCC, Clang and the Intel compiler. This gives us measurements for all microbenchmarks without any interference with the data race detection tools. We also collect runtime and memory consumption results for all microbenchmarks with the data race detection tools.

We run each evaluation five times for every microbenchmark with the exact same input and independent of any other run. This is necessary because analysis results can differ dependent on the thread schedule and as reported in [12] this is necessary to get good analysis results.

The average values of runtime and memory consumption from the 5 independent runs are also used in the runtime behavior analysis. To simplify the analysis, we select only a data size of 1024 for microbenchmarks in the var-length set.

We conduct an empirical analysis using the collected data and focus on the following four tasks:

1. Analyzing the differences between microbenchmarks in DataRaceBench

2. Comparing selected data race detection tools
3. Investigating differences between specific tools
4. Correlation study using data analysis tool

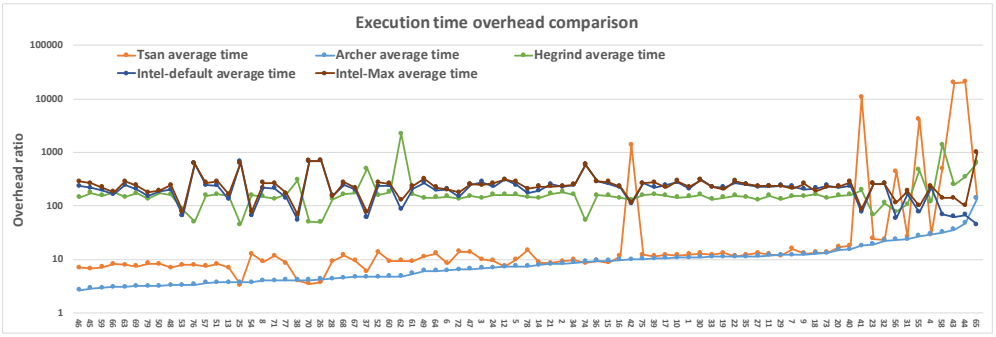
Differences within DataRaceBench DataRaceBench was designed to represent OpenMP programs used for data race detection. There are 70 microbenchmarks with a single OpenMP parallel region, 6 microbenchmarks with multiple OpenMP parallel regions (#41 - #44, #55, #56), and 3 microbenchmarks with no OpenMP parallel region (#25, #26, #70). In the results we discovered that the number of OpenMP threads used in the test has no impact on the execution time and memory consumption for microbenchmarks #25, #26, #70, #74, and #76. The microbenchmarks #25, #26, and #70 have the SIMD property label and have only a `OMP SIMD` pragma in the microbenchmark. Without `omp parallel` used in test codes, these tests will be run by only a single thread. The microbenchmarks #74 and #76 have the `num_threads(10)` clause to enforce having 10 OpenMP threads during the execution. The microbenchmarks #25, #26, and #70 also show the lowest execution time and memory consumption, with and without tools involved, in the sorted experiment results, and are run with a single thread and therefore have no OpenMP runtime overhead involved. The selected tools can perform data race detection only in multi-threaded programs (with OpenMP or Pthread). Therefore, the tools cannot detect data races existent in SIMD loops.

Microbenchmarks consuming high testing time are mainly from the Polyhedral microbenchmarks (#41 - #44, #55, #56), programs with nested loops and an OpenMP loop as their innermost loop (#37, #38, #62), programs with multidimensional arrays (#23, #31, #32, #37, #38, #58), and a program with high loop iterations (#65). Based on the results we find that microbenchmarks with large arrays (#4, #15, #31, #32, #37, #38) all have higher memory consumption in the experiments with all data race detection tools.

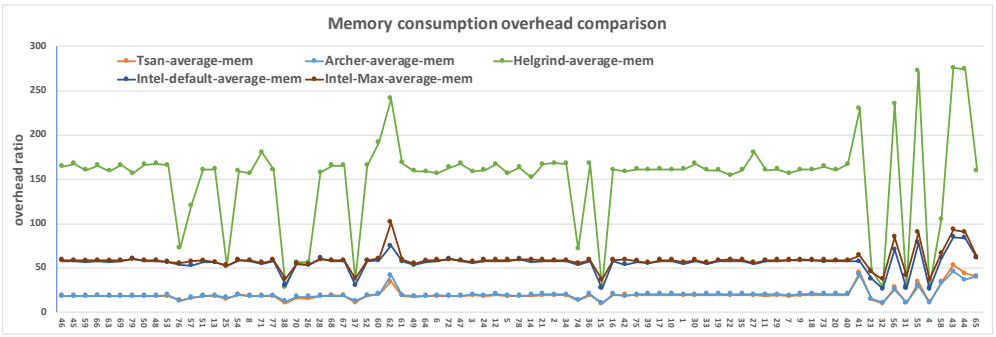
Comparing data race detection tools We observe the number of OpenMP threads used in experiments has a strong impact on both runtime and memory consumption. Experiments with more OpenMP threads use additional memory and have longer execution time. In this section we downselect the data to show only results from tests using 72 OpenMP threads, matching the hardware thread number available in the testing node. The computation in the program, the design of data race detection tool, and the software and hardware environment - all have an influence on the runtime and memory consumption in the data race detection.

To effectively compare all the selected tools, we choose the overhead ratio, the ratio between overhead (in time or memory consumption) and the corresponding baseline measurement, as the metric in the comparison. Figure 1a and Figure 1b show the overhead ratio in execution time and memory consumption from all tool configurations. The X axis in figures presents the IDs of all microbenchmarks in a sorted order based on the time overhead ratio from Archer. As Archer has the lowest overhead ratio for most of the microbenchmarks, the sorted order of Archer's result shows a clear referencing line in the figure. Archer is also the tool with highest accuracy from previous experiments in [12] and experiments in this paper.

There are two distinct groups shown in the runtime overhead comparison: (1) ThreadSanitizer and Archer have an overhead ratio around or lower than 10 for most



(a) Execution time comparison



(b) Memory consumption comparison

Fig. 1: Tool comparison

microbenchmarks; (2) Helgrind and Intel Inspector with two different configurations have a higher overhead ratio, between 100 to 1000, in most of the microbenchmarks. Although ThreadSanitizer and Archer are based on the same tool and show a low overhead ratio in most cases, we find ThreadSanitizer has the highest overhead ratio among all selected tools for the six Polyhedral microbenchmarks (#41 - #44, #55, #56). The figure shows a high similarity between the results from two configurations of Intel Inspector. The max-resource configuration shows a higher overhead ratio in 4 Polyhedral microbenchmarks (#43, #44 #55, #56), a microbenchmark with Jacobi kernel (#58), and a microbenchmark with a high OpenMP loop iteration count (#65).

The memory consumption comparison shows 3 different groups in Figure 1b. Archer and ThreadSanitizer have a very similar overhead ratio (around 20) for all microbenchmarks. Intel Inspector also has a high similarity in the overhead ratio (between 55 to 60) with two different configurations. Helgrind has the highest memory overhead ratio, most microbenchmarks around 160, but for 6 microbenchmarks higher than 220.

Comparing all results regardless of the OpenMP thread number used in the experiments, ThreadSanitizer and Archer have a time overhead ratio lower than 100, Helgrind has a range between 50 and 500, Inspector has a range at 200 to 2000. Regarding the memory consumption overhead: ThreadSanitizer and Archer have a range between 10 to 20, Helgrind has the highest number with a range between 50 and 250, Intel Inspector has a range at 20 to 80. From our collected data, we find that using a

higher number of OpenMP threads results in a higher execution time, but does not always result in a higher time overhead ratio. On the other hand, having more OpenMP threads, always leads to a high memory consumption and a higher memory consumption ratio. The time overhead is less sensitive to the OpenMP thread number used in the experiments. Comparing the highest time overhead ratio to the lowest time overhead ratio for each microbenchmark using different number of OpenMP threads, we see ThreadSanitizer has an average of 1.84×, Archer has 1.99×, Helgrind has 4.68×, Intel has 1.77× and 1.65× for default and max-resource configuration. Doing the same comparison for the memory consumption overhead ratio, ThreadSanitizer has an average of 4.62×, Archers has 2.94×, Helgrind has 7.77×, Intel has 7.96× and 5.96× for default and max-resource configuration.

Comparing ThreadSanitizer and Archer We further investigate the differences between ThreadSanitizer and Archer, especially for those microbenchmarks with a high difference in execution time between these two tools. Archer adopts static polyhedral analysis to detect data dependencies within a test code. A detected OpenMP parallelizable loop (i.e. a loop that is found to have no conflicting dependencies, such that it can be run in parallel) will be black-listed in Archer for dynamic analysis checking. This can reduce the testing time for Archer. For comparison we also use modified Archer command line arguments to perform data race detection without static analysis provided from LLVM's Polly. For all 79 microbenchmarks, using Archer with static analysis has a range between 0.66× to 1.09×, or an average of 0.97×, of execution time using Archer without static analysis. As Archer does not report the parallelizable loops detected by its static analysis, we do not have information whether loops are detected and blacklisted for the dynamic analysis. Microbenchmarks in DataRaceBench that were generated using polyhedral transformations are the candidates to be detected by the polyhedral analysis in LLVM's Polly. We exam the results from Polyhedral microbenchmarks in the DataRaceBench and the execution time using Archer with static analysis is 0.83×, on average, of the execution time from experiments using Archer without static analysis. We can conclude that the static analysis in Archer can help reduce the runtime of the data race detection if parallelizable loops are detected.

ThreadSanitizer has an extremely high time overhead ratio for polyhedral microbenchmarks (#43 - #44 #55, #56) and microbenchmark with the Jacobi kernel(#58). We use the CPU profiler tool from gperftools (originally Google Performance Tools)[7] to inspect two selected microbenchmarks, adi-tile-no.c (#44) and jacobikernel-orig-no.c (#58). CPU profiler uses a default frequency of 100 interrupts per second to profile active CPU usage. The profiler does not consider idle time (sleeping, blocked on locks, waiting for IO, waiting for work etc) in its report. The profile report can generate an annotated call graph with timing information and top N functions sorted by the execution time. Table 6 and Table 7 list the top 3 functions and total sample counts from the profile reports. For adi-tile-no.c (#44), ThreadSanitizer and Archer have 1518 and 1067 total sample counts respectively. This implies that ThreadSanitizer has 1.42× more active CPU usage than Archer. However, the execution time report shows that ThreadSanitizer (1474.2 sec.) and Archer (3.4 sec.) has a 432.9× difference in execution time. For jacobikernel-orig-no.c (#58), ThreadSanitizer has

[7] `https://github.com/gperftools/gperftools`

7.17× more active CPU usage than Archer. The execution times are 121.3 seconds for ThreadSanitizer and 7.8 seconds for Archer (15.51× difference).

The function call `kmp_flag_64::wait` is found in the top 3 functions for both microbenchmarks using both tools. It contains spin wait loop that does pause, then yield, and sleep in OpenMP runtime and it is likely to appear at a synchronization point. A `__kmp_release` from another thread has to appear to wake up this function from sleep. As idle time is not profiled by CPU profiler, this `kmp_flag_64::wait` could contribute a significant amount of sleeping time in the execution. We observe that ThreadSanitizer finishes reporting detected data races to stdout in the very beginning of the testing and then goes into an idle mode till the end of execution. The OpenMP standard specifies several high-level synchronization points: barrier, critical, atomic, taskwait, single, task, and reduction. ThreadSanitizer lacks information about these OpenMP synchronization points. In contrast, Archer with its OpenMP-awareness, has the advantage to use the annotation API of ThreadSanitizer to mark synchronization points within the OpenMP runtime and potentially save testing time for in the synchronization points. Our analysis concludes that ThreadSanitizer can require more time than Archer at the OpenMP synchronization points in the data race detection.

	ThreadSanitizer			Archer		
functions	Name	Count	%	Name	Count	%
#1	kmp_flag_64::wait	457	30.1	kmp_flag_64::wait	334	31.3
#2	__GI___sched_yield	222	14.6	__GI___sched_yield	291	27.3
#3	__kmp_hardware_timestamp	208	13.7	__kmp_hardware_timestamp	162	15.2
Total count	1518			1067		

Table 6: Profile report for adi-tile-no.c

	ThreadSanitizer			Archer		
functions	Name	Count	%	Name	Count	%
#1	kmp_flag_64::wait	15218	28.1	__tsan_read8	2337	30.9
#2	__GI___sched_yield	10410	19.2	kmp_flag_64::wait	1430	18.9
#3	__tsan::ReportRace	9499	17.5	__tsan_read4	1017	13.5
Total count	54173			7554		

Table 7: Profile report for jacobikernel-orig-no.c

Comparing default option and max-resource option in Intel Inspector The default data race detection configuration in Intel Inspector, ti2 analysis, aims to find out if a data race exists. In the configuration with maximum resources, we use the ti3 analysis which tries to answer where are the data races in the program. We setup the analysis knob to use an extreme scope to detect data races on the stack with a 1-byte data race detection granularity and cross-thread stack access detection. An additional analysis knob, use-maximum-resources, is set to allow Intel Inspector to exploit maximum system resources in data race detection. The collected data does not show significant difference between the results of the two configurations. There are higher variations in time overhead ratio and memory consumption overhead ratio for

microbenchmarks with more computation involved, i.e. polyhedral microbenchmarks and Jacobi kernels. Many microbenchmarks in DataRaceBench were designed to be compact and representative for OpenMP programs, but might not have enough complexity in the computation. Therefore, we do not see a significant difference between the runtime behaviors from Intel with default and maximum resource configurations. We do expect to see the configuration with maximum resources consuming more memory and execution time for real-world applications.

ThreadSanitizer		Archer		Helgrind		Intel-default		Intel-Max	
Time	Memory	Time	Memory	Time	Memory	Time	Memory	Time	Memory
6 (0.48)	4 (0.85)	8 (0.27)	4 (0.86)	6 (0.25)	4 (0.92)	8 (0.24)	4 (0.91)	8 (0.35)	4 (0.88)
8 (0.30)	6 (0.58)	3 (0.16)	6 (0.50)	3 (0.20)	6 (0.15)	4 (0.22)	7 (0.24)	4 (0.31)	7 (0.35)
3 (0.27)	7 (0.30)	4 (0.10)	7 (0.27)	5 (0.08)	7 (0.07)	6 (0.05)	6 (0.13)	3 (0.15)	6 (0.08)

Table 8: Correlation report with following attribute ID and correlation value. Attribute list: (1) tool, (2) filename, (3) haverace, (4) threads, (5) dataset, (6) races, (7) elapsed-time, (8) used-mem.

Comparing data race detection tools We use the data analysis tool, Weka[7], to investigate the correlations within the collected data. There are eight attributes used by the data analysis tool representing the eight columns of data recorded in the csv files. We can calculate the correlation between each attribute using Weka and rank the attributes with the strongest correlation. Table 8 shows the top three attributes with strong correlation to the execution time and memory consumption using the selected data race detection tools. Each cell in the table shows the attribute ID with its correlation value. Higher correlation values represent stronger correlation between attributes. The data analysis report shows that the thread number (attribute #4) has a strong correlation to memory consumption for every tool. The number of races detected has a moderate correlation to the memory consumption for Archer and ThreadSanitizer. The report shows no attribute with a strong correlation to the execution time for all tools. Only the number of races detected also has a moderate correlation, 0.48, to the execution time for ThreadSanitizer.

5 Related Work

All of the tools evaluated in this paper are dynamic; they run the target program under instrumentation and analyze the execution trace [29]. Many dynamic analyses use a happens-before approach. Reads and writes to shared memory are modeled by a partial order over events within the system [11]. This technique is heavily dependent on the application scheduler, and may miss many latent races. Many advances have been made in this area over the years by using more specialized concepts than traditional vector clocks in order to reduce overhead [5,4], expanding it to single-threaded event-driven programs [14], and defining additional relations such as casually-precedes [25].

Lockset analyses such as Eraser [22] present an alternative to happens-before techniques; they infer the set of mutually-exclusive locks that protect each shared location. If a variable's lockset is empty then accesses to that location may trigger races. These

analyses can find races that happens-before techniques cannot, but they incur steep performance costs.

Hybrid approaches combining both methods have also been developed [29,19,16,8,24]. These methods leverage information about local control flow, recent access, and common race patterns in order to dynamically adjust the analysis. This leads to greater flexibility when balancing accuracy and performance, as well as enabling long-term [29] and large-scale [24] analyses that might not be possible with other techniques.

Static data race detection techniques do not require the program to be executed in order to identify data-races. Static tools do not rely on instrumented schedulers, and therefore may find races that dynamic tools could not. Locksmith [20] is one such tool that seeks to correlate locks with the shared memory locations they guard. It over-approximates the set of data races, possibly returning some false positives. Another analysis seeks to improve the detection of shared variables [9] by performing pointer analysis in order to find global variables that are locally aliased. The RELAY analysis [27] modularizes each source of unsoundness in its analysis so that more accurate methods can be substituted when they are developed. OmpVerify [2] is a static race detector that targets OpenMP exclusively. It uses a polyhedral model to determine data dependencies in shared data.

An analysis of Intel Thread Checker was performed in 2008 [10], evaluating its performance in detecting races during loop parallel and section parallel codes. The benchmark suite used for the evaluation was not released along with the paper. Another work evaluated three data race detection tools: Sun's Thread Analyzer, Intel's Thread Checker and GNU's RaceStand [6]. With the EPCC OpenMP benchmark, Thread Analyzer was about five times faster than Thread Checker and two times slower than RaceStand.

Similar multi-tool analyses have been performed with other languages. Two targeting the Java language [1,28] analyzed several data race detection tools and compared the accuracy and performance of each. The first [1] compared RaceFuzzer, RacerAJ, JCHORD, Race Condition Checker, and Java RaceFinder. The authors compared the compilation time, accuracy, precision, along with several other metrics. Java RaceFinder performed the best on their tests, although it only reported the first race found even if there were others in the program.

The second [28] focused on detection methods rather than tools, and compared five different algorithms: FastTrack, Acculock, Multilock-HB, SimpleLock+, and casually precedes (CP) detection. The report used FastTrack as a baseline to compare detection accuracy and performance against. Multilock-HB reported the most races without any false-positives, but generated significant overhead; SimpleLock+ was had the lowest overhead, but missed at least one race that MultiLock found.

There exist several frameworks that can simplify the evaluation of large sets of programs: DataMill deals with the problem of varying hardware [17], EMP addresses the problem of varying runtimes on repeated executions of the same program [26], and BenchExec [3] provides a framework for execution (measurement and control), collecting data from large benchmark sets, and results representation (tables, graphs). As the complexity of the DataRaceBench benchmarks grows and their tested environments become more complicated we consider using one of those frameworks in future.

6 Conclusion

In this paper, we present a runtime and memory evaluation for four data race detection tools using the DataRaceBench suite. Regardless of the effectiveness in finding data races, ThreadSanitizer and Archer (based on ThreadSanitizer), have the lowest overheads in time and memory consumption among the four tools. These two tools have high similarity in the overhead of memory consumption but differ more in the time overhead for several microbenchmarks. We conclude the differences in runtime are from two factors: (1) the static analysis available in Archer blacklists serial execution and uses polyhedral analysis to determine whether affine loops nests have a data race to reduce dynamic analysis testing. (2) ThreadSanitizer can have a higher idling time for spin-wait at the synchronization points compared to Archer. The OpenMP-awareness in Archer provides higher accuracy in performing data race detection in OpenMP programs. The time overhead ratio from Helgrind in data race detection is between the overheads from Intel Inspector and ThreadSanitizer-based tools. Helgrind has the highest memory consumption overhead ratio among the selected tools. The reason is that Helgrind lacks knowledge about OpenMP semantics and reports many false positive data races from microbenchmarks in DataRaceBench. Intel Inspector offers many configurable parameters to increase the accuracy and provides detailed information about detected data races. The two configurations used in our evaluation, default and maximum resources, show only a small difference in many microbenchmarks in DataRaceBench. A few microbenchmarks, those with multiple OpenMP parallel regions and more computation involved in the parallel loops, show higher memory consumption for the configuration with maximum resources. Intel Inspector also stores the tracing information to disk and it allows users to use an interface to review and inspect the detected races with references to the source code. Intel Inspector delivers higher accuracy in data race detection when configured with maximum resources verse its default configuration. Through data analysis, we find a strong correlation between the number of OpenMP threads and memory consumption. There are weak correlations between the runtime and other data collected in the experiments.

In conclusion, Archer and Intel Inspector with OpenMP awareness provide higher accuracy in finding data races. Intel Inspector, with two configurations, has a roughly 4 times higher overhead in memory consumption and a much higher overhead in execution time compared to Archer. Intel Inspector requires additional disk storage to store tracing information and writing data to disk could have an impact on the overall runtime.

References

1. Alowibdi, J.S., Stenneth, L.: An empirical study of data race detector tools. In: 2013 25th Chinese Control and Decision Conference (CCDC). pp. 3951–3955 (May 2013). https://doi.org/10.1109/CCDC.2013.6561640, http://dx.doi.org/10.1109/CCDC.2013.6561640
2. Basupalli, V., Yuki, T., Rajopadhye, S.V., Morvan, A., Derrien, S., Quinton, P., Wonnacott, D.: ompverify: Polyhedral analysis for the openmp programmer. In: Chapman, B.M., Gropp, W.D., Kumaran, K., Müller, M.S. (eds.) OpenMP in the Petascale Era - 7th International Workshop on OpenMP, IWOMP 2011, Chicago, IL, USA, June 13-15, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6665, pp. 37–53. Springer (2011). https://doi.org/10.1007/978-3-642-21487-5_4, http://dx.doi.org/10.1007/978-3-642-21487-5_4
3. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. International Journal on Software Tools for Technology Transfer pp. 1–29 (Nov 2017). https://doi.org/10.1007/s10009-017-0469-y, https://doi.org/10.1007/s10009-017-0469-y
4. Effinger-Dean, L., Lucia, B., Ceze, L., Grossman, D., Boehm, H.: Ifrit: interference-free regions for dynamic data-race detection. In: Leavens, G.T., Dwyer, M.B. (eds.) Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012. pp. 467–484. ACM (2012). https://doi.org/10.1145/2384616.2384650, http://doi.acm.org/10.1145/2384616.2384650
5. Flanagan, C., Freund, S.N.: Fasttrack: efficient and precise dynamic race detection. In: Hind, M., Diwan, A. (eds.) Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009. pp. 121–133. ACM (2009). https://doi.org/10.1145/1542476.1542490, http://doi.acm.org/10.1145/1542476.1542490
6. Ha, O.K., Kim, Y.J., Kang, M.H., Jun, Y.K.: Empirical comparison of race detection tools for openmp programs. In: Grid and Distributed Computing. pp. 108–116. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
7. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The weka data mining software: An update. SIGKDD Explor. Newsl. **11**(1), 10–18 (Nov 2009)
8. Huang, J., Meredith, P.O., Rosu, G.: Maximal sound predictive race detection with control flow abstraction. In: O'Boyle, M.F.P., Pingali, K. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014. pp. 337–348. ACM (2014). https://doi.org/10.1145/2594291.2594315, http://doi.acm.org/10.1145/2594291.2594315
9. Kahlon, V., Yang, Y., Sankaranarayanan, S., Gupta, A.: Fast and accurate static data-race detection for concurrent programs. In: Damm, W., Hermanns, H. (eds.) Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4590, pp. 226–239. Springer (2007). https://doi.org/10.1007/978-3-540-73368-3_26
10. Kim, Y., Kim, D., Jun, Y.: An empirical analysis of intel thread checker for detecting races in openmp programs. In: Lee, R.Y. (ed.) 7th IEEE/ACIS International Conference on Computer and Information Science, IEEE/ACIS ICIS 2008, 14-16 May 2008, Portland, Oregon, USA. pp. 409–414. IEEE Computer Society (2008). https://doi.org/10.1109/ICIS.2008.79, http://dx.doi.org/10.1109/ICIS.2008.79
11. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978). https://doi.org/10.1145/359545.359563, http://doi.acm.org/10.1145/359545.359563

12. Liao, C., Lin, P.H., Asplund, J., Schordan, M., Karlin, I.: 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. pp. 11:1–11:14. SC '17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3126908.3126958, http://doi.acm.org/10.1145/3126908.3126958
13. Liao, C., Quinlan, D.J., Willcock, J.J., Panas, T.: Semantic-aware automatic parallelization of modern applications using high-level abstractions. International Journal of Parallel Programming **38**(5), 361–378 (2010)
14. Maiya, P., Kanade, A., Majumdar, R.: Race detection for android applications. In: O'Boyle, M.F.P., Pingali, K. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014. pp. 316–325. ACM (2014). https://doi.org/10.1145/2594291.2594311, http://doi.acm.org/10.1145/2594291.2594311
15. Müehlenfeld, A., Wotawa, F.: Fault detection in multi-threaded c++ server applications. In: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 142–143. PPoPP '07, ACM, New York, NY, USA (2007)
16. O'Callahan, R., Choi, J.: Hybrid dynamic data race detection. In: Eigenmann, R., Rinard, M.C. (eds.) Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2003, June 11-13, 2003, San Diego, CA, USA. pp. 167–178. ACM (2003). https://doi.org/10.1145/781498.781528, http://doi.acm.org/10.1145/781498.781528
17. de Oliveira, A.B., Petkovich, J.C., Reidemeister, T., Fischmeister, S.: Datamill: Rigorous performance evaluation made easy. In: Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering. pp. 137–148. ICPE '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2479871.2479892, http://doi.acm.org/10.1145/2479871.2479892
18. Petersen, P., Shah, S.: Openmp support in the intel® thread checker. In: International Workshop on OpenMP Applications and Tools. pp. 1–12. Springer (2003)
19. Poznianski, E., Schuster, A.: Efficient on-the-fly data race detection in multithreaded C++ programs. In: 17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France, CD-ROM/Abstracts Proceedings. p. 287. IEEE Computer Society (2003). https://doi.org/10.1109/IPDPS.2003.1213513, http://dx.doi.org/10.1109/IPDPS.2003.1213513
20. Pratikakis, P., Foster, J.S., Hicks, M.W.: LOCKSMITH: context-sensitive correlation analysis for race detection. In: Schwartzbach, M.I., Ball, T. (eds.) Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006. pp. 320–331. ACM (2006). https://doi.org/10.1145/1133981.1134019, http://doi.acm.org/10.1145/1133981.1134019
21. Sack, P., Bliss, B.E., Ma, Z., Petersen, P., Torrellas, J.: Accurate and efficient filtering for the intel thread checker race detector. In: Proceedings of the 1st workshop on Architectural and system support for improving software dependability. pp. 34–41. ACM (2006)
22. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.E.: Eraser: A dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. **15**(4), 391–411 (1997). https://doi.org/10.1145/265924.265927, http://doi.acm.org/10.1145/265924.265927
23. Serebryany, K., Iskhodzhanov, T.: Threadsanitizer: Data race detection in practice. In: Proceedings of the Workshop on Binary Instrumentation and Applications. pp. 62–71. WBIA '09, ACM, New York, NY, USA (2009)

24. Serebryany, K., Iskhodzhanov, T.: Threadsanitizer: data race detection in practice. In: Proceedings of the Workshop on Binary Instrumentation and Applications. pp. 62–71. ACM (2009)
25. Smaragdakis, Y., Evans, J., Sadowski, C., Yi, J., Flanagan, C.: Sound predictive race detection in polynomial time. In: Field, J., Hicks, M. (eds.) Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012. pp. 387–400. ACM (2012). https://doi.org/10.1145/2103656.2103702, http://doi.acm.org/10.1145/2103656.2103702
26. Suh, Y., Snodgrass, R.T., Kececioglu, J.D., Downey, P.J., Maier, R.S., Yi, C.: EMP: execution time measurement protocol for compute-bound programs. Softw., Pract. Exper. **47**(4), 559–597 (2017). https://doi.org/10.1002/spe.2476, https://doi.org/10.1002/spe.2476
27. Voung, J.W., Jhala, R., Lerner, S.: RELAY: static race detection on millions of lines of code. In: Crnkovic, I., Bertolino, A. (eds.) Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007. pp. 205–214. ACM (2007). https://doi.org/10.1145/1287624.1287654, http://doi.acm.org/10.1145/1287624.1287654
28. Yu, M., Park, S.M., Chun, I., Bae, D.H.: Experimental performance comparison of dynamic data race detection techniques. ETRI Journal **39**(1), 124–134 (02 2017). https://doi.org/10.4218/etrij.17.0115.1027, http://dx.doi.org/10.4218/etrij.17.0115.1027
29. Yu, Y., Rodeheffer, T., Chen, W.: Racetrack: efficient detection of data race conditions via adaptive tracking. In: Herbert, A., Birman, K.P. (eds.) Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005. pp. 221–234. ACM (2005). https://doi.org/10.1145/1095810.1095832, http://doi.acm.org/10.1145/1095810.1095832