Exploring Regression of Data Race Detection Tools Using DataRaceBench

Pei-Hung Lin, Chunhua Liao, Markus Schordan, Ian Karlin
Lawrence Livermore National Laboratory, Livermore, CA 94550, USA
Email: {lin32, liao6, schordan1, karlin1}@llnl.gov

***Abstract*—DataRaceBench is an OpenMP benchmark suite designed to systematically and quantitatively evaluate data race detection tools. It has been used by several research and development groups to measure the quality of their tools. In this paper we explore how to evaluate the regression of data race detection tools in the presence of observed tool errors. We define how to generating consistent, reproducible, and comparable evaluation results and a detailed evaluation process with a set of configuration and execution rules. We also outline differences in the evaluation of dynamic and static data race detection tools. In addition to the evaluation results, we explore and suggest different ways to process and present the data, with a focus on tool errors. Using DataRaceBench we show an accuracy regression for several popular data race detection tools in recent release cycles.**

***Index Terms*—Data race detection, Benchmark, OpenMP**

I. Introduction

DataRaceBench [15] is a dedicated OpenMP benchmark suite to evaluate data race detection tools. The goal of this benchmark suite is two-fold: (1) to capture requirements for data race detection in OpenMP programs, and (2) to assess the status of current data race detection tools. The initial release (v.1.0.1) of DataRaceBench contained 72 microbenchmarks written in C99. The latest release (v1.2.0) has improved of the OpenMP 4.5 specification with a total of 116 microbenchmarks [16]. 59 of these microbenchmarks, also called race-yes programs, have known data races. Each microbenchmark contains exactly one pair of read/write or write/write code locations that cause a data race. The other 57 race-no microbenchmarks are data race free. A subset of the microbenchmarks support variable length arrays to allow configurable input data sizes.

Since its release, DataRaceBench has been used by several research groups to evaluate data race detection tools [3], [9], [20]. However, DataRaceBench does not contain sufficient guidelines for users to generate consistent results. For example, it is not clear how to calculate a tool's precision, recall, and accuracy when there are random results and compile-time or runtime errors. In addition, users have modified some of the tests, cherry-picked a subset, or interpreted results in different ways.

In this paper, we explore how to consistently evaluate different data race detection tools and generate reproducible, comparable results. The contributions of this paper include:

- Systematically studying the key factors impacting a consistent evaluation of data race detection tools;
- Proposing solutions to improve the consistency and reproducibility of tool evaluation;
- Suggesting different ways to process and present the resulting data;
- Evaluating the regression of several data race detection tools in the presence of observed tool errors. A regression happens when a change to a piece of software causes new faults and/or re-emergence of old faults. If applied consistently, DataRaceBench can be used as a regression test suite to identify such problems.
- Releasing improved evaluation results which are automatically generated by our new evaluation process.

II. Overview of Consistent Evaluation

To improve consistency in tool evaluation, all steps in an evaluation process should be consistent other than the variable being tested. In the context of data race detection tool evaluation, we identify several key factors, as shown in Table I.

TABLE I
Factors impacting consistent evaluation of tools

Factor	Description	Solution
Benchmarks	Test programs and inputs used	DataRaceBench
Platforms	Hardware, OS and software	Standard VMs
Tools	Tool versions, configuration and installation	Docker Images
Process	The steps and settings used	Automated scripts
Results	Processing and interpreting results	Standard metrics
Randomness	Random results reported	Reporting ranges
Errors	Compile or runtime errors	Adding error rate metrics

We elaborate on the factors in the following list:

- Benchmarks. We should use a commonly defined, complete set of benchmark programs to conduct the evaluation experiments. DataRaceBench is an example for this purpose.
- Platforms. The machines used for evaluation play an important role of getting consistent results. Ideally, identical platforms with the same hardware and software configurations should be used for experiments. However, often only platforms with similar configurations can be found. With the increasing popularity of cloud machines, the community can leverage commonly used virtual machine instances to improve platform consistency.

- Tools. Tools can be installed and configured in many different ways. A tool often depends on several other software packages. We should clearly define the versions, installation and execution configurations for all related software packages. In addition, containers such as Docker may be used to package various software components when applicable.
- Evaluation process. We use scripts to automate the evaluation process as much as possible, including both experiments and processing of data. This helps eliminate ambiguity and reduce human errors.
- Processing and interpretation of results. Tools may use terms differently and or generate results in different formats. We should define key terms clearly and unify the presentation of results.
- Randomness. Multi-threaded programs often have non-deterministic behaviors depending on the number of threads, the compiler, and hardware platforms. We report the range of metrics instead of a single value to capture the variation among different program executions.
- Errors. It is common to encounter errors during the evaluation process. The errors may come from the tools, supportive compilers, or runtime libraries. It must be clear how these errors are represented in the evaluation results.

III. Terminology and Metrics

The traditional definition of a data race is as follows [24]:

" A *data race* can occur when two concurrent threads access a shared variable and when at least one access is a write, and the threads use no explicit mechanism to prevent the accesses from being simultaneous. "

Tool regression testing is the repeated testing of an already tested tool, after modification, to discover any defects introduced or uncovered as a result of the changes/updates to the tool. In our use case we evaluate different versions of released data race detection tools with DataRaceBench and compare the results to detect a regression, i.e. that one of the computed scores and/or metrics is lower than for a previous tool version.

Similar to our evaluation in [15] we compute the usual test metrics for data races as follows. When a data race is detected it is considered a 'positive' test outcome, if no data race is detected a 'negative' outcome. If the outcome is as expected (known for all the benchmarks) then it is reported as a 'true' case, otherwise a 'false' case. The combination gives four possible cases, based on which different metrics are defined.

Let TP be the set of True Positives, FP the set of False Positives, TN be set of True Negatives, and FN be the set of False Negatives, then we can define the following four metrics:

- Precision $P = \frac{|TP|}{|TP \cup FP|}$
- Recall $R = \frac{|TP|}{|TP \cup FN|}$
- Accuracy $A = \frac{|TP \cup TN|}{|TP \cup TN \cup FP \cup FN|}$

The precision metric (also called positive predictive value) measures the ratio of true positives to the sum of true positives and false positives. It reflects the confidence that a reported positive by a tool is a real one. Recall (also known as sensitivity) is a measure of a tool's ability to find true positives out of the sum of true positives and false negatives. Accuracy gives the chance of having correct reports out of all positive and false reports generated by a tool. For all the three metrics, higher values are better.

A tool error can be considered to be different than a reported wrong result or treated in a similar way. This impacts how results about the tool are reported, in particular it may change the metrics values and consequently the selection of the respective tool. In our evaluation we exclude tool errors (e.g. seg-faults and reported unsupported features) from the computation of metrics, and also report a success-rate. Thus, we consider a ratio of successful tests and failing tests in addition to the usual metrics.

Alternatively one could add the failing test cases to the false positive/negative cases, giving less favorable results for a tool, but this can be misleading, because the tool did actually not report the existence or absence of a data race. Since we also consider that multiple tools can be combined and used together, preference is given to metrics for non-failing tool executions. Therefore we exclude tool failures from the metrics computation, but do include them in the success-rate in the evaluation report.

IV. Experiments

A. Tools Selected

We selected four dynamic data race detection tools and one static data race detection tool for evaluation and regression testing. All the selected tools, dynamic and static, support data race detection for OpenMP programs.

ThreadSanitizer 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. ThreadSanitizer was not developed for OpenMP parallel programs. However, we have found that ThreadSanitizer can effectively support data race detection for OpenMP codes when using the LLVM OpenMP runtime configured with the `LIBOMP_TSAN_SUPPORT` turned on.

Archer [2] is an OpenMP data race detector that exploits ThreadSanitizer [8] to achieve scalable happens-before tracking. It uses static analysis to reduce false positives generated by the dynamic analysis performed by ThreadSanitizer. Archer has to use the LLVM/Clang compiler which ThreadSanitizer is based on to compile the benchmarks. In our experiments, Archer uses the LLVM OpenMP Runtime with OMPT support provided. Two available versions of Archer, built on ThreadSanitizer version 3.9.1 and 6.0.0 respectively, are used in the experiments.

Intel Inspector [11] is a dynamic analysis tool that detects threading and memory errors in C, C++ and Fortran codes. It can work with compilers other than Intel's. However, it is recommended to use Intel's OpenMP runtime with Inspector to avoid incorrect threading error analysis. We use the 2018 and 2919 versions of Intel Inspector, combined with four different Intel compiler versions, 17.0.2, 18.0.2, 19.0.0, and 19.0.4,

in our experiments. Intel Inspector provides different levels of analysis with the various configurations. We configure the maximum level for the analysis in our evaluation using the command line: `inspxe-cl -collect ti3 -knob scope=extreme -knob stack-depth=16 -knob use-maximum-resources=true`. We use the default OpenMP runtime provided with the Intel compiler in this experiment.

ROMP[1] is a recently released dynamic data race detection tool for OpenMP programs [9]. Using a hybrid algorithm combining happens-before analysis and lock set analysis, ROMP focuses on logical concurrency instead of runtime thread concurrency to reduce false negatives for tests which are sensitive to the number of threads. ROMP requires OpenMP runtime support be provided by LLVM.

DRACO is a static data race detection tool based on polyhedral data dependence analysis [30] and can prove that a program is data race free. Therefore the results have to be interpreted differently than dynamic analysis results. The tool also reports when it cannot verify the data race freedom of a program because of unsupported OpenMP features or code patterns, in which case it reports "unknown". When unknown is reported the program may have or may not have a data race. OpenMP runtime support is not needed for DRACO because its static analysis based data race detection does not execute the test program.

B. Hardware Platform

Our evaluation platform is the Quartz cluster hosted at the Livermore Computing Center [18]. Each computation node of the cluster has two Intel 18-core Xeon E5-2695 v4 processors with hyper threading enabled.

C. Evaluation Reports

We used Archer (2 versions), Intel Inspector (2 versions with 4 different compilers), ThreadSanitizer (4 versions), ROMP, and DRACO for the evaluation. To evaluate the dynamic tools we used the all 36 available hardware threads on the evaluation platform, and two data set sizes, 32 and 256, for the benchmarks allowing variable data sizes. We ran each test 3 times using the test harness script provided in DataRaceBench to catch mixed results produced by dynamic tools. A ten minutes limit is set for each test. The static tool is evaluated using only the default data set size, 64, for benchmarks allowing variable data size. Since static analysis is performed at the compile time, the analysis is only performed once for each benchmark.

During the evaluation we collect the test result, memory consumption and execution time. Since this paper focuses on the regression of data race detection tools, we do not analyze memory consumption and only report basic runtime data.

Table II shows the statistics of the evaluation for all dynamic tools. Each dynamic tool should perform a total number of 396=3*(100+16*2) tests, with 3 runs for 100 benchmarks each using fixed length arrays, and 16 benchmarks using variable length arrays with 2 data sizes each. Any lower number of reported successful tests is due to a missing executable, caused by compile time segmentation fault or an unsupported compiler feature. The static tool, DRACO, performs a total of 116 analysis runs, one for each benchmark. The counts of four possible test result sets, TP, FN, TN and FP, are reported and the three metrics are calculated according to the formula defined in section III. We collect two types of testing errors and two types of runtime testing errors in the statistics. The compile time errors include compile-time segmentation faults (CSF) and unsupported OpenMP features reported by compilers (CUN). There will be no executable generated with these two errors, therefore no test can be performed. Two types of runtime errors are collected in the statistics: execution crash at runtime (RSF) and execution exceeding timeout limit (RTO). Total testing time for the entire suite is reported in Table II for all the tests including those with runtime timeout and runtime execution crash.

In Table III we show the three evaluation metrics for precision, recall, and accuracy for all 116 micro-benchmarks of DataRaceBench. In this table only the error-free evaluation runs are included. Evaluation runs that crashed or had a time-out are excluded. The metrics are computed following the metrics reported in [15]. All test results (from all repeated runs with all the thread numbers and data set sizes used in the evaluation) associated with a benchmark are grouped together to determine the test result for the same benchmark. For example, we label a tool reporting TP for a benchmark only when all the associated test runs tested by the tool report TP. A mixed result (TP mixed with FN or TN mixed with FP) is labeled if different results are reported by all associated test runs. Treating a mixed result to be a true or false result introduces different metrics values. The min and max values reported in Table III shows the impact of the mixed results in the evaluation. In the best case scenario where a tool always reports consistent results and no mixed results are reported, the min and max values for a metric should be identical.

D. Dynamic Tools' Regression Results

In Table II, we show our regression evaluation for Archer, Intel Inspector, and ThreadSanitizer. [2] No regression results are available yet for ROMP since there is only one version of ROMP available. The results count each test run individually. In Table III and Table V the results count each micro-benchmark individually. We can see that over time the success-rate for Archer and Intel Inspector increases, but ThreadSanitizer's success rate decreased with its most recent release. In contrast, the metrics show a very different picture, with some metrics decreasing for every tool with more than one released version. (e.g. IntelInspector2018-Intel9.0.4 to IntelInspector2019-Intel9.0.4 (Recall); Tsan6.0.1 to Tsan7.1.0 to Tsan8.0.1 (Accuracy).

[1]https://github.com/zygyz/romp, commit # 6a0ad6d

[2]The Archer group suggests setting environment variable TSAN_OPTIONS to "ignore_noninstrumented_modules=1" for Archer 2.0 to avoid many false positive reported in OpenMP runtime.

TABLE II
EVALUATION STATISTICS: COUNTS ARE BASED ON ALL TEST INSTANCES
(SF: SEGMENTATION FAULT; UN: COMPILER UNSUPPORTED; TO: EXCEEDED TIMEOUT)

Tool-Compiler	**Successful Tests**	**Test Results**				**Metrics**			**Testing Error**				**Test Time (HH:MM:SS)**
		TP	FN	TN	FP	Precision	Recall	Accuracy	CSF	CUN	RSF	RTO	
Archer1.0-Clang3.9.1	376	187	24	145	0	1.00	0.89	0.93	5	5	10	0	00:06:11
Archer2.0-Clang6.0.0	386	202	20	156	3	0.99	0.91	0.94	0	5	0	0	00:06:17
Inspector2008-Intel17.0.2	392	195	30	156	9	0.96	0.87	0.90	2	0	0	0	01:32:50
Inspector2008-Intel18.0.2	396	198	27	160	8	0.96	0.88	0.91	0	0	0	3	02:04:34
Inspector2018-Intel19.0.0	396	213	12	60	108	0.66	0.95	0.69	0	0	0	3	03:41:17
Inspector2018-Intel19.0.4	396	198	27	160	11	0.95	0.88	0.90	0	0	0	0	01:33:54
Inspector2019-Intel17.0.2	392	195	30	159	6	0.97	0.87	0.91	2	0	0	0	01:37:08
Inspector2019-Intel18.0.2	396	195	30	162	6	0.97	0.87	0.91	0	0	0	3	02:04:49
Inspector2019-Intel19.0.0	396	214	11	61	107	0.67	0.95	0.70	0	0	0	3	03:32:55
Inspector2019-Intel19.0.4	396	195	30	164	7	0.97	0.87	0.91	0	0	0	0	01:37:27
ROMP-Clang8.0.0	384	198	18	144	6	0.97	0.92	0.93	0	6	9	3	00:59:20
Tsan5.0.2-Clang5.0.2	386	192	30	153	3	0.98	0.86	0.91	0	5	0	3	00:36:28
Tsan6.0.1-Clang6.0.1	386	195	27	156	3	0.98	0.88	0.92	0	5	0	0	00:07:34
Tsan7.1.0-Clang7.1.0	386	193	29	154	5	0.97	0.87	0.91	0	5	0	0	00:07:19
Tsan8.0.1-Clang8.0.1	384	184	38	152	4	0.98	0.83	0.89	0	6	0	0	00:07:03

TABLE III
METRICS FOR THE TOOLS: COUNTS ARE BASED ON THE NUMBER OF BENCHMARKS (MULTIPLE INSTANCES OF THE SAME BENCHMARK ARE COUNTED AS ONE)

Tool	Correctness Success Rate	Precision Range	Recall Range	Accuracy Range
Archer1.0-Clang3.9.1	0.80	1.00-1.00	0.85-0.89	0.92-0.94
Archer2.0-Clang6.0.0	0.90	0.98-0.98	0.90-0.91	0.94-0.95
Inspector2018-Intel17.0.2	0.87	0.93-0.96	0.85-0.88	0.89-0.92
Inspector2018-Intel18.0.2	0.90	0.94-0.96	0.86-0.90	0.90-0.93
Inspector2018-Intel19.0.0	0.66	0.61-0.61	0.95-0.95	0.66-0.66
Inspector2018-Intel19.0.4	0.90	0.93-0.95	0.86-0.92	0.90-0.93
Inspector2019-Intel17.0.2	0.90	0.96-0.96	0.86-0.86	0.91-0.91
Inspector2019-Intel18.0.2	0.91	0.96-0.96	0.86-0.86	0.91-0.91
Inspector2019-Intel19.0.0	0.66	0.61-0.62	0.95-0.97	0.66-0.68
Inspector2019-Intel19.0.4	0.91	0.94-0.96	0.86-0.86	0.91-0.91
ROMP-Clang8.0.0	0.85	0.96-0.96	0.91-0.91	0.93-0.93
Tsan5.0.2-Clang5.0.2	0.84	0.98-0.98	0.79-0.91	0.88-0.95
Tsan6.0.1-Clang6.0.1	0.86	0.98-0.98	0.83-0.91	0.90-0.95
Tsan7.1.0-Clang7.1.0	0.84	0.96-0.98	0.79-0.91	0.87-0.95
Tsan8.0.1-Clang8.0.1	0.81	0.96-0.98	0.76-0.86	0.85-0.92
DRACO	0.40	0.58-0.58	0.55-0.55	0.59-0.59

For dynamic tools, the number of threads and the number of runs influences the probability to detect a data race. With 36 threads we get always at least two expected results out of three runs for all benchmarks except two. In contrast, with just two threads, this is not the case, even with three runs a data race can be missed. However, there is no guarantee that a run creates a thread schedule revealing a data race. Therefore, the more runs the more likely it is to detected a data race. Essentially we want to create a high probability of different thread schedules. Not surprisingly, the maximum number of hardware threads is the most important factor, but for a few benchmarks we also notice that 2-3 repeated runs are (at least) necessary (with dynamic tools) until an existing data race is detected. This is true for those benchmarks where not every thread schedule causes a data race.

Table IV lists benchmarks that have compiler error, runtime error, or false data race detection reported by any selected combination of tools and compilers. The ID column shows the micro-benchmark ID given in the DataRaceBench Suite and the R column presents whether a program contains a data race. The abbreviated label in each column shows the tool and compiler combination with the version information: Arch. = Archer; Ins. = Inspector; Tsan = ThreadSanitizer; Cl. = Clang; In. = Intel compiler. The labels, CSF, CUN, RSF, and RTO, categorize the reported errors. The ✓and ✗symbols represent if the tool together with the selected compiler correctly reports data races.

Several reasons of failed data race detection for the first 72 micro-benchmarks were reported in [15]. Those include the case that a data race exists only when a specific number of OpenMP threads are used (in micro-benchmarks #6, #7 and #8) and when a data race exists in the SIMD programs (in micro-benchmarks #24 and #25). This evaluation report focuses on the regression of data race detection tools and the micro-benchmarks added into after DataRaceBench version 1.0.1.

Table IV shows the compilers failing with a segmentation-fault or an unsupported OpenMP feature of a given micro-benchmark. We find compile time segmentation faults only with the earliest version of Clang/LLVM (3.9.1) and the earliest version of Intel compiler (17.0.2). The trend of compiler development improving OpenMP support can be seen in the table.

From the report we find various versions of the Clang/LLVM compiler reporting unsupported OpenMP features for the same benchmarks. In contrast, the latest ThreadSanitizer with Clang/LLVM 8.0.x, reports one additional compile time error when compared to all previous versions (see Table V). In Table IV we can see that Clang/LLVM 8.0.x (used by ROMP and ThreadSanitizer 8.0.1) failed compiling micro-benchmark #97, DRB097-target-teams-distribute-orig-no.c.

Table IV also shows benchmarks that have mixed results, TP mixed with FN, or FN mixed with TN, and the respective tools and compilers. Mixed results indicate that multiple runs are necessary to get the expected answer.

TABLE IV
BENCHMARKS WITH COMPILER/TOOL REGRESSION
COLUMN ID: NUMBER OF BENCHMARK; COLUMN R: WHETHER A PROGRAM CONTAINS A DATA RACE
(✓CORRECT; ✗INCORRECT; CSF: COMPILER SEGMENTATION FAULT; CUN : COMPILER UNSUPPORTED
RSF: RUNTIME SEGMENTATION FAULT; RTO : RUNTIME TIMEOUT)

ID	R	Tool-Compiler														
		Arch.1-Cl.391	Arch.2-Cl.600	Ins.18-In.1702	Ins.18-In.1802	Ins.18-In.1900	Ins.18-In.1904	Ins.19-In.1702	Ins.19-In.1802	Ins.19-In.1900	Ins.19-In.1904	ROMP-Cl.800	Tsan5-Cl.502	Tsan6-Cl.601	Tsan7-Cl.710	Tsan8-Cl.801
5	Y	✓	✓	✗	✗	✓	✓/✗	✗	✗	✓	✗	✓	✓	✓	✓	✗
6	Y	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓/✗	✓/✗	✓/✗	✗
7	Y	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
8	Y	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
13	Y	✓	✗	✗	✓/✗	✗	✓/✗	✗	✗	✓/✗	✗	✓	✓/✗	✓	✓/✗	✗
23	Y	✓/✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗
24	Y	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
25	Y	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
27	Y	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
29	Y	✓	✓	✓/✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
34	Y	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓/✗	✓/✗	✓/✗	✓
37	Y	RSF	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
39	Y	✓	✓	✗	✓/✗	✓	✗	✗	✗	✓	✗	✓	✓	✓	✓	✓
40	Y	✓	✓	✓/✗	✗	✓	✓/✗	✗	✗	✓	✗	✓	✓/✗	✓/✗	✓/✗	✓/✗
41	N	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
42	N	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
43	N	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
44	N	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
45	N	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
46	N	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
47	N	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✗	✓	✓	✓	✓
48	N	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
49	N	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
50	N	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
52	N	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
53	N	RSF	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
54	N	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
55	N	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
56	N	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
57	N	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
59	N	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
60	N	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
61	N	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
62	N	RSF	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
63	N	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
64	N	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
65	N	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	RTO	✓	✓	✓	✓
66	N	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
67	N	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
68	N	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
71	N	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
72	N	✓	✓	✓/✗	✓/✗	✓	✓/✗	✓	✓	✓	✓/✗	✓	✓	✓	✓/✗	✓/✗
75	Y	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓/✗	✓	✓	✓
80	Y	✓/✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
82	Y	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓/✗	✓/✗	✓/✗	✓/✗
85	N	CSF	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓
86	Y	CSF	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓/✗	✓/✗
87	Y	CSF	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓/✗	✓/✗	✓/✗	✓/✗
91	N	CSF	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓
93	N	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
94	N	CUN	CUN	CSF	RSF	RSF	✓	CSF	RSF	RSF	✓	CUN	CUN	CUN	CUN	CUN
95	Y	CUN	CUN	✓	✓	✓	✓	✓	✓	✓	✓	CUN	CUN	CUN	CUN	CUN
96	N	CUN	CUN	✓/✗	✓	✗	✓	✓	✓	✗	✓	CUN	CUN	CUN	CUN	CUN
97	N	RSF	✗	✓	✗	✓	✓	✓	✗	✓	CUN	RTO	✓	✓	CUN	
99	N	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
100	N	CUN	CUN	✓	✓	✓	✓	✓	✓	✓	✓	CUN	CUN	CUN	CUN	CUN
102	N	CSF	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
105	N	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	RSF	✓	✓	✓	✓
106	Y	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	RSF	✓	✓	✓	✓
107	N	✓	✓	✗	✗	✗	✓	✗	✗	✓/✗	✓	✓	✓	✓	✓	✓
108	N	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓
110	N	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✗	✗	✗	✗
112	N	CUN	CUN	CSF	✓	✗	✓	CSF	✓	✗	✓	CUN	CUN	CUN	CUN	CUN
113	N	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
114	Y	✓	✓/✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓/✗	✓	✓	✓/✗
116	Y	RSF	✓	✓	✓	✓	✓	✓	✓	✓	✓	RSF	✓	✓	✓	✓

TABLE V
REGRESSION SUMMARY: COUNTS BASED ON BENCHMARKS

Tool-compiler	Benchmarks with Race						Benchmarks without Race						Compiler Error		Runtime Error	
	TP		TP/FN		FN		TN		TN/FP		FP					
Archer1.0-Clang3.9.1	46	39.66%	2	1.72%	6	5.17%	47	40.52%	0	0.00%	0	0.00%	10	8.62%	5	4.31%
Archer2.0-Clang6.0.0	52	44.83%	1	0.86%	5	4.31%	52	44.83%	0	0.00%	1	0.86%	5	4.31%	0	0.00%
Inspector2018-Intel17.0.2	50	43.10%	2	1.72%	7	6.03%	51	43.97%	2	1.72%	2	1.72%	2	1.72%	0	0.00%
Inspector2018-Intel18.0.2	51	43.97%	2	1.72%	6	5.17%	53	45.69%	1	0.86%	2	1.72%	0	0.00%	1	0.86%
Inspector2018-Intel19.0.0	56	48.28%	0	0.00%	3	2.59%	20	17.24%	0	0.00%	36	31.03%	0	0.00%	1	0.86%
Inspector2018-Intel19.0.4	51	43.97%	3	2.59%	5	4.31%	53	45.69%	1	0.86%	3	2.59%	0	0.00%	0	0.00%
Inspector2019-Intel17.0.2	51	43.97%	0	0.00%	8	6.90%	53	45.69%	0	0.00%	2	1.72%	2	1.72%	0	0.00%
Inspector2019-Intel18.0.2	51	43.97%	0	0.00%	8	6.90%	54	46.55%	0	0.00%	2	1.72%	0	0.00%	1	0.86%
Inspector2019-Intel19.0.0	56	48.28%	1	0.86%	2	1.72%	20	17.24%	1	0.86%	35	30.17%	0	0.00%	1	0.86%
Inspector2019-Intel19.0.4	51	43.97%	0	0.00%	8	6.90%	54	46.55%	1	0.86%	2	1.72%	0	0.00%	0	0.00%
ROMP-Clang8.0.0	51	43.97%	0	0.00%	5	4.31%	48	41.38%	0	0.00%	2	1.72%	6	5.17%	4	3.45%
Tsan5.0.2-Clang5.0.2	46	39.66%	7	6.03%	5	4.31%	51	43.97%	0	0.00%	1	0.86%	5	4.31%	1	0.86%
Tsan6.0.1-Clang6.0.1	48	41.38%	5	4.31%	5	4.31%	52	44.83%	0	0.00%	1	0.86%	5	4.31%	0	0.00%
Tsan7.1.0-Clang7.1.0	46	39.66%	7	6.03%	5	4.31%	51	43.97%	1	0.86%	1	0.86%	5	4.31%	0	0.00%
Tsan8.0.1-Clang8.0.1	44	37.93%	6	5.17%	8	6.90%	50	43.10%	1	0.86%	1	0.86%	6	5.17%	0	0.00%

E. Static Verification Tool Evaluation

DRACO was chosen as the data race verification to use in our evaluation [30]. DRACO can verify that a given polyhedral OpenMP for-loop is data race free. Verification tools usually report a 3-valued result: the specification or property could be verified, falsified, or it could not be determined either of the two cases. Reasons for definitive answer include, the tool ran out of resources or certain language constructs are not supported. Static analysis tools usually perform a may or must analysis and report a 2-valued result, where one of the results is either an over-approximation (may) or under-approximation (must). In this paper we only consider static verification tools with 3-valued results.

TABLE VI
DRACO EVALUATION RESULT WITH 3-VALUED RESULTS INCLUDING TRUE UNKNOWN (TU) AND FALSE UNKNOWN (FU).

Total	TP	TN	FP	FN	TU	FU	Error	Verification Time
116	26	20	0	0	28	33	9	00:14:17

DRACO is a static verification tool that reports three kinds of results: it has detected a data race that definitely exists, it has determined that there is definitely no data race in a given polyhedral parallel loop, or 'unknown'. It reports 'unknown' if any unsupported OpenMP directives are detected or the for-loop nest is not a polyhedral loop nest (i.e. it contains non-affine index accesses to array elements). The results for the 116 micro-benchmarks are shown in Table VI. For each of the two groups of micro-benchmarks with and without data races, there are two root-causes of reported 'unknown': (i) the tool determines that it cannot analyze the respective parallel loop (i.e. it is not an affine loop nest), (ii) it detected an unsupported OpenMP construct somewhere in the code. DRACO does not *report* false positives or false negatives, but it does report 'unknown' in all cases where it cannot determine a definitive answer. From a user perspective this is helpful, because it is not necessary to hunt down false-positives, but it would not be a fair comparison to compute the metrics with zero false positives/negatives. With regard to the metrics, we consider therefore the 'True Unknown' (TU) as false positives and the 'False Unknown' (FU) as false negatives. This gives scores as shown in Table III and allows to compare the static verification tool with other testing tools (otherwise the metrics would all be 1.0). However, the advantage in comparison to testing tools of proving data race freedom for 20 benchmarks is not represented in the metrics.

V. RELATED WORK

A. Dynamic data race detection tools

All the dynamic tools evaluated in this paper run the target program after instrumentation and analyze the execution trace [32]. 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 [14]. This technique is heavily dependent on the application scheduler, and may miss many latent races. Advances made in this area include using specialized concepts rather than traditional vector clocks in order to reduce overhead [5], [6], expanding it to single-threaded event-driven programs [19], and defining additional relations such as casually-precedes [27].

Lockset analyses such as Eraser [25] 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 [10], [21], [22], [26], [32]. 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 [32] and large-scale [26] analyses that might not be possible with other techniques.

B. Static data race detection tools

Static data race detection techniques do not require program execution to identify data-races. Static tools do not rely on

instrumented schedulers, and therefore may find races that dynamic tools could not. Locksmith [23] 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 [12] by performing pointer analysis in order to find global variables that are locally aliased. The RELAY analysis [29] modularizes each source of unsoundness in its analysis so that more accurate methods can be substituted when they are developed. OmpVerify [4] 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 [13], 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.

Similar multi-tool analyses have been performed with other languages. Two targeting the Java language [1], [31] 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 [31] 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.

C. Evaluation of data race detection tools

More recently, DataRaceBench [15] was created to systematically and quantitatively evaluate data race detection tools for OpenMP programs. The selected four tools being evaluated demonstrated different strengths and limitations for precision, recall and accuracy. Later, DataRaceBench was extended to cover more OpenMP 4.5 features. The enhanced version exposed more compiler and tool limitations related to OpenMP tasking and other newer OpenMP features [16]. Lin et. al. [17] also conducted runtime and memory evaluation of data race detection tools. DataRaceBench can be incorporated into any existing regression testing frameworks (e.g. Jenkins [28]) as concrete test cases.

JBench [7] is a collection of 48 JAVA programs and 985 data races. The details of data races are provided and the data races are classified into four categories: variable type, code structure, method span and cause. Three JAVA data race detectors were evaluated with JBench and over than 40% of false negative rate and more than $200\times$ overhead were reported. DataRaceBench and JBench both have a known number of data races and the data race information in the source codes. They can be effectively used to evaluate OpenMP and JAVA data race detection tools respectively.

VI. Conclusion

Our evaluation of multiple versions of four dynamic data race detection tools and one static data race detection tool identified regressions in the most recent releases of popular industry tools ThreadSanitizer and Intel Inspector. The research tool Archer also shows a regression. Although the success rate of passing micro-benchmarks is increasing for most tools' version transitions, more detailed metrics show significant changes. For example the recent Intel Inspector 2019 release shows a degradation in recall from 0.95 to 0.87 (see Table II).

The other two evaluated tools, ROMP a promising new dynamic data race detection tool, and DRACO a static verification tool under development, were presented recently and have not had a version transition yet, but are interesting to discuss in the context of comparative tool evaluations.

With multiple runs different results were reported for about 6% of the micro-benchmarks for ThreadSanitizer, and up to 3.5% for Intel Inspector. The most recent versions of Archer and ROMP do not show any difference in results with multiple runs.

In the future, we will add Fortran tests, support OpenMP 5.0 features, and group the micro-benchmarks based on OpenMP features and code characteristics for which we can see a correlation to regression results. We will also further investigate the combination of tools, in particular the combination of static and dynamic tools appear promising in cases where they only one tool can determine a definitive answer.

Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 and was supported by the LLNL-LDRD Program under Project No. 17-ERD-023. LLNL-CONF-787979.

References

[1] J. S. Alowibdi and L. Stenneth. An empirical study of data race detector tools. In *2013 25th Chinese Control and Decision Conference (CCDC)*, pages 3951–3955, May 2013.

[2] Simone Atzeni, Ganesh Gopalakrishnan, Zvonimir Rakamaric, Dong H Ahn, Ignacio Laguna, Martin Schulz, Gregory L Lee, Joachim Protze, and Matthias S Müller. ARCHER: effectively spotting data races in large OpenMP applications. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 53–62. IEEE, 2016.

[3] Simone Atzeni, Ganesh Gopalakrishnan, Zvonimir Rakamaric, Ignacio Laguna, Gregory L Lee, and Dong H Ahn. Sword: A bounded memory-overhead detector of OpenMP data races in production runs. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 845–854. IEEE, 2018.

[4] V. Basupalli, Tomofumi Yuki, Sanjay V. Rajopadhye, Antoine Morvan, Steven Derrien, Patrice Quinton, and David Wonnacott. ompverify: Polyhedral analysis for the OpenMP programmer. In Barbara M. Chapman, William D. Gropp, Kalyan Kumaran, and Matthias S. Müller, editors, *OpenMP in the Petascale Era - 7th International Workshop on OpenMP, IWOMP 2011, Chicago, IL, USA, June 13-15, 2011. Proceedings*, volume 6665 of *Lecture Notes in Computer Science*, pages 37–53. Springer, 2011.

[5] Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-Juergen Boehm. Ifrit: interference-free regions for dynamic data-race detection. In Gary T. Leavens and Matthew B. Dwyer, editors, *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*, pages 467–484. ACM, 2012.

[6] Cormac Flanagan and Stephen N. Freund. Fasttrack: efficient and precise dynamic race detection. In Michael Hind and Amer Diwan, editors, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 121–133. ACM, 2009.

[7] Jian Gao, Xin Yang, Yu Jiang, Han Liu, Weiliang Ying, and Xian Zhang. Jbench: A dataset of data races for concurrency testing. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, pages 6–9, New York, NY, USA, 2018. ACM.

[8] Google. Threadsanitizer, 2017.

[9] Yizi Gu and John Mellor-Crummey. Dynamic data race detection for OpenMP programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 61. IEEE Press, 2018.

[10] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. Maximal sound predictive race detection with control flow abstraction. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 337–348. ACM, 2014.

[11] Intel. Intel inspector 2017, 2017.

[12] Vineet Kahlon, Yu Yang, Sriram Sankaranarayanan, and Aarti Gupta. Fast and accurate static data-race detection for concurrent programs. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 226–239. Springer, 2007.

[13] Young-Joo Kim, Daeyoung Kim, and Yong-Kee Jun. An empirical analysis of intel thread checker for detecting races in OpenMP programs. In Roger Y. Lee, editor, *7th IEEE/ACIS International Conference on Computer and Information Science, IEEE/ACIS ICIS 2008, 14-16 May 2008, Portland, Oregon, USA*, pages 409–414. IEEE Computer Society, 2008.

[14] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[15] Chunhua Liao, Pei-Hung Lin, Joshua Asplund, Markus Schordan, and Ian Karlin. DataRaceBench: a benchmark suite for systematic evaluation of data race detection tools. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 11. ACM, 2017.

[16] Chunhua Liao, Pei-Hung Lin, Markus Schordan, and Ian Karlin. A semantics-driven approach to improving DataRaceBench's OpenMP standard coverage. In *International Workshop on OpenMP*, pages 189–202. Springer, 2018.

[17] Pei-Hung Lin, Chunhua Liao, Markus Schordan, and Ian Karlin. Runtime and memory evaluation of data race detection tools. In *International Symposium on Leveraging Applications of Formal Methods*, pages 179–196. Springer, 2018.

[18] LLNL. Livermore computing quartz system, 2017.

[19] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. Race detection for android applications. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 316–325. ACM, 2014.

[20] Hassan Salehe Matar and Didem Unat. Runtime determinacy race detection for OpenMP tasks. In *European Conference on Parallel Processing*, pages 31–45. Springer, 2018.

[21] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In Rudolf Eigenmann and Martin C. Rinard, editors, *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2003, June 11-13, 2003, San Diego, CA, USA*, pages 167–178. ACM, 2003.

[22] Eli Poznianski and Assaf Schuster. 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*, page 287. IEEE Computer Society, 2003.

[23] Polyvios Pratikakis, Jeffrey S. Foster, and Michael W. Hicks. LOCKSMITH: context-sensitive correlation analysis for race detection. In Michael I. Schwartzbach and Thomas Ball, editors, *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, pages 320–331. ACM, 2006.

[24] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[25] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[26] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, pages 62–71. ACM, 2009.

[27] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. Sound predictive race detection in polynomial time. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 387–400. ACM, 2012.

[28] John Ferguson Smart. *Jenkins: The Definitive Guide: Continuous Integration for the Masses*. " O'Reilly Media, Inc.", 2011.

[29] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. RELAY: static race detection on millions of lines of code. In Ivica Crnkovic and Antonia Bertolino, editors, *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*, pages 205–214. ACM, 2007.

[30] F. Ye, M. Schordan, C. Liao, P. Lin, I. Karlin, and V. Sarkar. Using polyhedral analysis to verify OpenMP applications are data race free. In *2018 IEEE/ACM 2nd International Workshop on Software Correctness for HPC Applications (Correctness)*, pages 42–50, Nov 2018.

[31] Misun Yu, Seung-Min Park, Ingeol Chun, and Doo-Hwan Bae. Experimental performance comparison of dynamic data race detection techniques. *ETRI Journal*, 39(1):124–134, 02 2017.

[32] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In Andrew Herbert and Kenneth P. Birman, editors, *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005*, pages 221–234. ACM, 2005.