LLNL-CONF-728717

DataRaceBench: A Benchmark Suite for Systematic Evaluation of Data Race Detection Tools

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

April 6, 2017

The International Conference for High Performance
Computing, Networking, Storage and Analysis
Denver, CO, United States
November 12, 2017 through November 17, 2017

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

DataRaceBench: A Benchmark Suite for Systematic Evaluation of Data Race Detection Tools

Chunhua Liao
Lawrence Livermore National Laboratory
Livermore, California, USA
liao6@llnl.gov

Pei-Hung Lin
Lawrence Livermore National Laboratory
Livermore, California, USA
lin32@llnl.gov

Joshua Asplund
Lawrence Livermore National Laboratory
Livermore, California, USA
asplund1@llnl.gov

Markus Schordan
Lawrence Livermore National Laboratory
Livermore, California, USA
schordan1@llnl.gov

Ian Karlin
Lawrence Livermore National Laboratory
Livermore, California, USA
karlin1@llnl.gov

ABSTRACT

Data races in multi-threaded parallel applications are notoriously damaging while extremely difficult to detect. Many tools have been developed to help programmers find data races. However, there is no dedicated OpenMP benchmark suite to systematically evaluate data race detection tools for their strengths and limitations.

In this paper, we present DataRaceBench, an open-source benchmark suite designed to systematically and quantitatively evaluate the effectiveness of data race detection tools. We focus on data race detection in programs written in OpenMP, the popular parallel programming model for multi-threaded applications. In particular, DataRaceBench includes a set of microbenchmark programs with or without data races. These microbenchmarks are either manually written, extracted from real scientific applications, or automatically generated optimization variants.

We also define several metrics to represent effectiveness and efficiency of data race detection tools. Using DataRaceBench and its metrics, we evaluate four different data race detection tools: Helgrind, ThreadSanitizer, Archer, and Intel Inspector. The evaluation results show that DataRaceBench is effective to provide comparable, quantitative results and discover strengths and weaknesses of the tools being evaluated.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**; **Correctness**; • **Computing methodologies → Parallel programming languages**;

KEYWORDS

Data Race Detection, OpenMP, Benchmark Suite

ACM Reference Format:
Chunhua Liao, Pei-Hung Lin, Joshua Asplund, Markus Schordan, and Ian Karlin. 2017. DataRaceBench: A Benchmark Suite for Systematic Evaluation of Data Race Detection Tools. In *Proceedings of SC17, Denver, CO, USA, November 12–17, 2017,* 14 pages.
https://doi.org/10.1145/3126908.3126958

1 INTRODUCTION

With the advent of many-core and GPU accelerated systems threaded programming models are being used in order to exploit on-node parallelism. One particularly hard-to-solve bug in these models is data races. Data race bugs result in non-deterministic code, which means they will not appear every time the code is run, making them notoriously difficult to detect and fix. 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.

To assist developers in detecting and fixing data races, tools have been developed (e.g. Intel Inspector [3] and ThreadSanitizer [6]). However, most of these tools [11, 12, 15, 28] are tested by leveraging benchmarks not designed for testing race detection tools (including SPECOMP [9], OMPSCR [18], NAS Parallel Benchmarks [22], Rodinia [16], etc.). Often customized tweaks are made to inject data races and the resulting evaluation are not systematic or reproducible. The lack of an apples-to-apples comparison of these tools makes it difficult for users to select the right tools for their problems. In particular it is not clear how many types of data races these tools can reliably and efficiently catch.

In other communities benchmarks are regularly used to measure and assess quality in a common, reproducible and systematic way. Good benchmarks help a community clarify problems to be solved, build common evaluation metrics, guide future development, and foster collaborations. Popular benchmarks, such as SPEC (Standard Performance Evaluation Corporation) [1] and LINPACK [17], play important roles in the HPC community for performance improvements. However, in the data race detection research field, there is no dedicated benchmark suite to collect representative input examples and generate comparable common evaluation metrics.

In this paper, we present an ongoing effort to build a comprehensive benchmark suite (named DataRaceBench) specifically designed for systematic evaluation of data race detection tools. Our initial suite focuses on OpenMP. While other threading programming models, such as Pthreads, Cilk and Intel Threading Building Blocks are also in use, OpenMP is by far the most popular API in the HPC community for writing multi-threaded programs. It is higher level than the previous models and has wider vendor adoption than other competitors, such as OpenACC. In a time of significant architectural diversity where portability and productivity are important these advantages have led to widespread adoption. In this paper, we close the benchmark gap and make the following contributions:

- We thoroughly analyze desired features of a dedicated benchmark suite for evaluating both static and dynamic data race detection tools. These features are related to coverage, scalability, parameterized execution, automation, accessibility, extensibility, correctness, and so on.
- We present a set of property labels to indicate different data race patterns, including unresolvable dependences, missing data sharing clauses, missing synchronization, undefined behaviors, and so on. These high-level labels apply to different base languages allowed in OpenMP. A concrete data race test program may be associated with one or more labels.
- We create a benchmark suite consisting of a set of small programs which can be used to evaluate data race detection tools and generate systematic and quantitative metrics.
- We use our benchmark suite to evaluate four representative state-of-the-art data detection tools: Helgrind, ThreadSanitizer, Archer, and Intel Inspector.
- Our results show DataRaceBench can be used to determine the strengths and limitations of the tools we tested.

The remainder of this paper is organized as follows. Section 2 clarifies some terms related to data race detection and our evaluation. Section 3 gives details about DataRaceBench's design considerations and current content. In Section 4, we select four representative data race detection tools and evaluate them using DataRaceBench. Finally, we describe related work in Section 5 and conclude our paper in Section 6.

2 TERMINOLOGY

The traditional definition of a data race is as follows [39]:

> 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.

A subset of data races, so called *benign data races*, are data races that have no potentially to be harmful. Any schedule of threads will produce a desired result, where the desired result may be a concrete number, an effect (e.g. to terminate all worker threads), or a specific state of the application. In the presented version of the benchmark suite we do not include benchmarks with benign data races as they can not be detected automatically without user annotations. Note even for a benign data race one might still want to ensure that it is indeed a benign data race, by verifying that such an annotated data race can indeed happen for some thread schedule. For example, a technique that provides support for classifying data races as benign by replaying different thread schedules has been presented in [30].

There exist two types of analysis tools. Dynamic analysis tools cannot prove the absence of data races, and when a dynamic analysis tool reports it has not found a data race, data races might still exist.

In contrast, a static software verification tool can prove the *absence of errors*. In particular, it can provide a three valued answer: it will either determine that an implementation of a given specification is correct, incorrect, or unknown (within provided resource constraints). For the purpose of data races the specification can be pre-defined in the dedicated data race detection tool and does not need to be provided by the user since it is independent of the application semantics. Note this is different to benign data races and so called high-level data races [8]. For that reason we focus on "traditional" data races in our benchmark suite and do not consider high-level data races and benign data races. Our benchmark suite also contains a set of benchmarks with no data races; those benchmarks are particularly interesting for software verification tools, as the evaluated dynamic tools cannot provide a definite answer to the question whether a given program is data race free.

If an existing data race is not detected by a dynamic tool, it may still exist. If an analysis tool reports a data race, but de-facto it does not exist, it is called a *false-positive (FP)* analysis result. If a (static) analysis tool reports that definitely no data race exists, but there actually does exist a data race in the benchmark, it is called a *false-negative (FN)* analysis result. If an analysis tool detects an existing data race it is called a *true-positive (TP)* and if an analysis tool determines that definitely no data race exists, and there is indeed no data race in the benchmark, it is called a *true-negative (TN)* analysis result.

For analysis tools it is common to define metrics for *precision (P)*, *recall (R)*, and accuracy (A) as follows: $P = TP/(TP + FP)$, $R = TP/(TP + FN)$, and $A = (TP + TN)/(TP + FP + TN + FN)$. The precision metric 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 is a measure of a tool's ability to find true positives out of the sum of true positives and false negatives. Finally, 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. We provide results for these metrics in our tool evaluation.

3 DATA RACE BENCHMARK SUITE

In this section we describe the design criteria and guidelines used during the development and collection of the microbenchmarks. We define property labels to categorize the microbenchmarks and give an overview of the origins from where we extracted the microbenchmarks.

3.1 Design Criteria and Our Solutions

The goal of our benchmark suite is two-fold: 1) to capture the requirements related to data race detection in OpenMP programs, and 2) to assess the status of current data race detection tools. In order to reach our goal, we first summarize a list of criteria for a good benchmark suite and then present our solutions, as shown in Table 1.

Representative: One major challenge in the design of a benchmark suite is making sure the benchmark suite sufficiently represents the features being evaluated. Our solution is to look at the entire OpenMP specification to extract representative correct and incorrect usage patterns related to data races. The extraction process includes reading literature for common mistakes [44], learning from existing microbenchmarks and regression tests of relevant

Criteria	Our Solutions
Representative	Target all OpenMP usage, with optimized versions
Scalable	Allow different data sizes and thread counts
General	Support both static and dynamic tools
Accessible	BSD open-source license, available from github.com
Extensible	Self-contained programs, easy to revise and add
Easy to use	Automated scripts for execution and report generation
Quantitative	Generate a range of standard metrics
Correct	Consistent with tool results and manual inspection

Table 1: Design criteria and solutions of DataRaceBench

tools, and creating simplified kernels from real applications. We also include optimization variants automatically generated by PolyOpt [33], a polyhedral optimization tool.

We use an iterative process by selecting a set of data race detection tools and checking if DataRaceBench can give significantly different results for these tools. A benchmark suite with good feature coverage should be able to easily identify the differences among tools being evaluated.

Scalable: As a benchmark suite for parallel computing, DataRaceBench is also set up to scale. DataRaceBench supports different data input sizes, varying number of threads, and multiple runs via parameterized execution. Reproducibility is achieved by providing scripts that perform parameterized evaluations.

General: The benchmark suite is designed to be generally applicable for both dynamic as well as static analysis tools. For dynamic analysis concrete inputs are provided in our microbenchmarks. For static analysis tools, microbenchmarks with arbitrary inputs (i.e. a wide range of possible inputs) exist.

Accessible: To enable maximal use of our benchmark suite, we decided to use the permissive BSD open-source license to allow both research and commercial usage. The benchmark suite is available at https://github.com/LLNL/dataracebench.

Extensible: Building a good benchmark suite requires enduring effort and community help. The language specifications and usage constantly evolve. DataRaceBench makes additions and changes easy. We organize our microbenchmark programs as self-contained programs, which can be easily modified. New programs can also be added with minimum dependences on the existing microbenchmark files.

Easy to use: Running microbenchmarks can be tedious and time-consuming. We provide configurable scripts to automatically evaluate one or multiple tools, across different execution parameters, including the number of threads, input data size, the number of runs, and so on. Our scripts also automatically generate final reports.

Quantitative: The results of the suite are quantitative metrics, including standard ones such as precision, recall, and accuracy. This enables an easy comparison of different tools or different versions of the same tool.

Correct: Microbenchmarks must be correct in the sense that they do not contain unintentional errors. We use a range of compilers (with all warning options turned on) and correctness tools to ensure the absence of unintentional errors in our microbenchmark programs. For example, Valgrind is used to capture memory leaks when executing the microbenchmarks.

We elaborate the current implementation details of DataRaceBench in the following subsections.

3.2 Design Guidelines

We have created a set of microbenchmarks to help evaluate data race detection tools. In order to enable flexible and easy use, several design guidelines are followed when creating these microbenchmarks. They include:

- Each microbenchmark should be as small as possible to represent typical data race detection use cases. For example, there are programs demonstrating the use of one or more OpenMP constructs and/or a common parallel computing pattern (for example, reduction, stencil, indirect array accesses, etc.).
- 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 program has a main function to support dynamic data race detection.
- To probe static data race detection tools, we include microbenchmarks that allow arbitrary problem sizes (i.e. the size of arrays on which computations are performed depend on input values).
- We want to categorize code patterns (or properties) of data races in OpenMP programs. Currently, all microbenchmarks are implemented in C99 to represent these patterns. The same patterns can be implemented in different languages (C++ or Fortran).
- The microbenchmarks are divided into two categories: microbenchmarks with known data races (named as the race-yes set) and those which are known to be data race free (named as the race-no set). A tool's data race detection results on the benchmark suite can be easily judged as true positive, true negative, false negative and false positive, which can be used to calculate and compare metrics like precision and recall.
- If possible, a program in the race-yes set, should only contain a single pair of source locations that cause data races. For static tools, this is used to check if they can catch the right number of location pairs causing data races. For dynamic tools, we can check if the tool consolidated multiple runtime data races caused by the same pair of source code locations, into one data race.
- All microbenchmark programs have builtin input data. To probe the impact of changing input data sizes, a subset of microbenchmarks with known data races are converted into one additional version using C99 variable-length arrays (called var-length set). The size of a variable-length array is specified by one command line option.

3.3 Property Labels

We assign property labels to the collected microbenchmarks, as shown in Table 2. These labels essentially indicate different patterns that we have found in OpenMP programs with or without data races. As a set of high-level properties, the same set of labels apply to different base languages allowed in OpenMP, including C, C++ and Fortran.

Property labels for race-yes set	Property labels for race-no set
Y1: Unresolvable dependences	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 2: Property labels of microbenchmarks

For the set of programs without data races, there are labels indicating programs that are embarrassingly parallel (plain OpenMP parallel for directives are sufficient to express their parallelism), those using various data sharing clauses (such as private, reduction, etc.), those using synchronization constructs (such as critical, atomic, barrier, locks, etc.), those using SIMD directives, those using accelerator directives, those using special language features (such as C99 restrict keyword), and finally those representing typical numerical patterns. Similarly, the set of programs with known data races have labels more or less mirroring those of the race-no set of programs. The differences are that some of the programs have unresolvable dependences, which are defined in this paper as loop carried dependences which cannot be eliminated by adding data sharing or synchronization OpenMP constructs. There exists also a label for programs with data races caused by undefined behaviors, such as out-of-bounds array accesses. For a microbenchmark program, it may be associated with multiple property labels.

3.4 Origins of Microbenchmarks

As shown in Table 3, multiple sources are used to ensure our microbenchmarks are representative and comprehensive. Some microbenchmark programs originate from a regression test set accumulated over several years for a tool named AutoPar [26], which automatically inserts OpenMP directives into serial input codes. This regression test set includes examples demonstrating correct or incorrect uses of OpenMP constructs as defined in the OpenMP specification. It also includes common and relevant mistakes collected from the literature [44]. The automatically generated OpenMP versions from the regression test set are the basis for the race-no set. Some race-no microbenchmarks are changed to create race-yes microbenchmarks, after removing necessary OpenMP clauses (such as private, reduction, etc.). AutoPar's regression tests also include programs which should not be parallelized due to loop carried dependences. These tests are used to create an additional race-yes set, after purposely adding wrong OpenMP directives.

Six of the microbenchmarks are parallel optimization variants of three algorithms from the PolyBench/C 4.2 benchmark suite [34]. PolyBench/C 4.2 is a collection of 30 benchmarks containing static control parts (subclasses of general loop nests that can be represented in the polyhedral model) and representing various application domains (linear algebra, image processing, physics simulation, dynamic programming, statistics, and etc.). We use PolyOpt, a polyhedral optimization tool, to generate parallel output variants for the benchmarks in PolyBench/C. The parallel optimization variants are generated with two optimization options in PolyOpt: one option parallelizes the code by performing a data dependence analysis using the polyhedral model and inserting OpenMP pragmas in the detected parallelizable loops. The other option enables fixed tiling and takes additional steps to perform loop tiling before the auto-parallelization and has the capability to detect vectorizable loops and insert SIMD pragmas for compiler vectorization. Currently, PolyOpt generates non-standard SIMD directives. We manually changed them in the microbenchmarks to be standard OpenMP SIMD directives.

Listing 1: Data races caused by out of bound access

```
  double b[n][m];
#pragma omp parallel for private(j)
  for (i=0; i<n; i++)
    for (j=0; j<m; j++)
      b[i][j]=b[i][j-1];
```

Microbenchmarks marked as "LLNL App" have been created to mimic data access patterns from development branches of real scientific applications developed at LLNL (the names of the applications are intentionally omitted). All reported data races have been fixed during development. Finally, we added brand new tests, such as the SIMD microbenchmark programs, to cover more OpenMP program patterns (indicated as property labels in this paper).

3.5 Examples

Due to space limitations, we only present some of the example microbenchmark programs in detail. Listing 1 shows a kernel demonstrating data races caused by an out-of-bounds access to a 2-D array. In this example, the outer level loop is parallelized using OpenMP. However, the inner level loop has out-of-bounds array accesses caused by $b[i][j-1]$ when j is equal to 0. For example, array element $b[2][0-1]$ becomes $b[1][3]$ due to linearized row-major storage of $b[n][m]$. This causes a loop-carried data dependence for the parallelized loop with the index variable i.

Listing 2 mimics a data access pattern from a real scientific application developed at LLNL. It represents a data race which manifests itself only if 36 or more threads are used. In this program, two pointers ($xa1$ and $xa3$) have a distance of 12 ($xa3 - xa1 = 12$). They are used as base addresses for indirect array accesses using another index set array ($indexSet[N]$). This index set array has two elements with distance 12 ($indexSet[5] - indexSet[0] = 533 - 521 = 12$). So there is a loop carried data dependence between the first and the sixth iteration (when i equals 0 or 5). The code's OpenMP loop uses the default scheduling (i.e. dividing the iterations into equally sized chunks and assigning at most one chunk to each thread). It is possible that the two dependent iterations will be scheduled within the same iteration chunk assigned to the same thread. So the occurrences of runtime data races depend on the OpenMP scheduling policies and/or the number of threads used. For a dynamic tool to reliably find this data race, iteration 0 and 5 must be scheduled to two different threads. In this example, one way is to use at least 36 threads ($180/36 = 5$ iterations per chunk).

`--polyopt-parallel-only`

`--polyopt-fixed-tiling`

Microbenchmarks with known data races			
Label(s)	**Program Name**	**Origin**	**Description**
Y1	antidep1	AutoPar	Anti-dependence within a single loop
Y1	antidep2	AutoPar	Anti-dependence within a two-level loop nest
Y7	indirectaccess1	LLNL App	Indirect access with overlapped index array elements
Y7	indirectaccess2	LLNL App	Overlapping index array elements when 36 or more threads are used
Y7	indirectaccess3	LLNL App	Overlapping index array elements when 60 or more threads are used
Y7	indirectaccess4	LLNL App	Overlapping index array elements when 180 or more threads are used
Y2	lastprivatemissing	AutoPar	Data race due to a missing lastprivate() clause
Y3	minusminus	AutoPar	Unprotected −− operation
Y3	nowait	AutoPar	Missing barrier due to a wrongfully used nowait
Y6	outofbounds	AutoPar	Out of bound access of the 2nd dimension of array
Y1	outputdep	AutoPar	Output dependence and true dependence within a loop
Y1	plusplus	AutoPar	++operation on array index variable
Y2	privatemissing	AutoPar	Missing private() for a temp variable
Y2	reductionmissing	AutoPar	Missing reduction() for a variable
Y3	sections1	New	Unprotected data writes in parallel sections
Y1,Y4	simdtruedep	New	SIMD instruction level data races
Y1,Y5	targetparallelfor	New	data races in loops offloaded to accelerators
Y3	taskdependmissing	New	Unprotected data writes in two tasks
Y1	truedep1	AutoPar	True data dependence among multiple array elements within a single level loop
Y1	truedepfirstdimension	AutoPar	True data dependence of first dimension for a 2-D array accesses
Y1	truedeplinear	AutoPar	Linear equation as array subscript
Y1	truedepscalar	AutoPar	True data dependence due to scalar
Y1	truedepseconddimension	AutoPar	True data dependence on 2nd dimension of a 2-D array accesses
Y1	truedepsingleelement	AutoPar	True data dependence due to a single array element

Microbenchmarks without known data races			
Label(s)	**Program Name**	**Origin**	**Description**
N2	3mm-parallel	Polyhedral	3-step matrix-matrix multiplication, non-optimized version
N2,N4	3mm-tile	Polyhedral	3-step matrix-matrix multiplication, with tiling and nested SIMD
N2	adi-parallel	Polyhedral	Alternating Direction Implicit solver, non-optimized version
N2,N4	adi-tile	Polyhedral	Alternating Direction Implicit solver, with tiling and nested SIMD
N1	doall1	AutoPar	Classic DOAll loop operating on a one dimensional array
N1	doall2	AutoPar	Classic DOAll loop operating on a two dimensional array
N1	doallchar	New	Classic DOALL loop operating on a character array
N2	firstprivate	AutoPar	Example use of firstprivate
N6	functionparameter	LLNL App	Arrays passed as function parameters
N6	fprintf	New	Use of fprintf()
N2	getthreadnum	New	single thread execution using `if(omp_get_thread_num()==0)`
N7	indirectaccesssharebase	LLNL App	Indirect array accesses using index arrays without overlapping
N1	inneronly1	AutoPar	Two-level nested loops, inner level is parallelizable. True dependence on outer level
N1	inneronly2	AutoPar	Two-level nested loops, inner level is parallelizable. Anti dependence on outer level
N7	jacobi2d-parallel	Polyhedral	Jacobi with array copying, no reduction, non-optimized version
N4,N7	jacobi2d-tile	Polyhedral	Jacobi with array copying, no reduction, with tiling and nested SIMD
N7	jacobiinitialize	AutoPar	The array initialization parallel loop in Jacobi
N7	jacobikernel	AutoPar	Parallel Jacobi stencil computation kernel with array copying and reduction
N2	lastprivate	AutoPar	Example use of lastprivate
N7	matrixmultiply	AutoPar	Classic i-k-j order matrix multiplication using OpenMP
N7	matrixvector1	AutoPar	Matrix-vector multiplication parallelized at the outer level loop
N7	matrixvector2	AutoPar	Matrix-vector multiplication parallelized at the inner level loop with reduction
N2	outeronly1	AutoPar	Two-level nested loops, outer level is parallelizable. True dependence on inner level
N2	outeronly2	AutoPar	Two-level nested loops, outer level is parallelizable. Anti dependence on inner level
N7	pireduction	AutoPar	PI calculation using reduction
N6	pointernoaliasing	LLNL App	Pointers assigned by different malloc calls, without aliasing
N6	restrictpointer1	LLNL App	C99 restrict pointers used for array initialization, no aliasing
N6	restrictpointer2	LLNL App	C99 restrict pointers used for array computation, no aliasing
N3	sectionslock1	New	OpenMP parallel sections with a lock to protect shared data writes
N1,N4	simd1	New	OpenMP SIMD directive to indicate vectorization of a loop
N1,N5	targetparallelfor	New	data races in loops offloaded to accelerators
N3	taskdep1	New	OpenMP task with depend clauses to avoid data races

Table 3: The list of microbenchmarks

Listing 2: data races when >= 36 threads are used

```
#define N 180
int indexSet[N] = {
//Note: indexSet[5]- indexSet[0] =
//      533 - 521 =  12
521, 523, 525, 527, 529, 533,
547, 549, 551, 553, 555, 557,
// omitted code here...
 };

#pragma omp parallel for
for(i=0; i< N; ++i)
{
  int idx=indexSet[i];
  xa1[idx]+=1.0;
  xa3[idx]+=3.0;
}
```

Listing 3: Parallel loop operating on a character array

```
char a[SIZE];
#pragma omp parallel for
for(i=0; i<100; i++)
  a[i]=a[i]+1;
```

Listing 4: Loop tiling with nested SIMD directives

```
void init_array(int n,double A[500][500],
                double B[500][500])
{
  int c1, c2, c3, c4;
  if(n>=1) {
#pragma omp parallel for private(c3, c4, c2)
    for(c1=0; c1<=(((n-1)*16<0?((16<0?-((-(n-1)
        +16+1)/16):-((-(n-1)+16-1)/16))):(n-1)/16)
        ); c1++) {
      for(c2=0; c2<=(((n-1)*16<0?((16<0?-((-(n-1)
          +16+1)/16):-((-(n-1)+16-1)/16))):(n-1)
          /16)); c2++) {
        for(c3=16*c2; c3<=((16*c2+15<n-1?16*c2+15:
            n-1)); c3++) {
#pragma omp simd
          for(c4=16*c1; c4<=((16*c1+15<n-1?16*c1
              +15:n-1)); c4++) {
            A[c4][c3]=(((double)c4)*(c3+2)+2)/n;
            B[c4][c3]=(((double)c4)*(c3+3)+3)/n;
          }
        }
      }
    }
  }
}
```

A counterpart microbenchmark program also exists (named as indirectaccesssharebase in Table 3). In this example, none of any pairs of indexSet's elements has a distance of 12 so $xa1$ and $xa3$ can never overlap. As a result, there are no data races in this program.

Listing 3 shows a kernel operating on an array of characters. This loop does not contain data races. However, for a dynamic tool monitoring memory accesses at 4-byte granularity, it may report a false positive since two consecutive iterations may access two characters falling into the same 4-byte memory block.

Finally, listing 4 shows a Jacobi initialization OpenMP kernel, from the jacobi2d-tile microbenchmark, with 16×16 loop tiling and nested SIMD, generated by PolyOpt and modified by us to use standard OpenMP SIMD directives. The generated expressions `n+-1` are replaced with `n-1` here in this text for better readability.

4 EVALUATION

In order to assess the effectiveness of our benchmark suite, we used DataRaceBench version 1.0.1 to evaluate data race detection tools. In this section, we describe the tools we selected and present our experimental results.

4.1 Data Race Detection Tools

Data race detection tools can be grouped into two categories: those using static analysis and those using dynamic analysis. In this paper, we evaluate four dynamic data race detection tools: Helgrind, ThreadSanitizer, Archer and Intel Inspector. Helgrind and ThreadSanitizer are recommended to be used for applications with a POSIX thread API. We include them in our evaluation to observe how well a race detection tool designed for a lower-level API can detect data races in higher-level OpenMP programs. The versions of the selected data race detection tools are listed in Table 4, with specific compilers used with these tools (either to build the tools or to compile the microbenchmarks, or both). For example, Archer (the towards_tr4 branch) and ThreadSanitizer are built using LLVM and clang version 4.0.1.

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 4: data race detection tools: versions and compilers

An evaluation with static analysis and verification tools is future work, as some of the interesting verification tools require code annotations (e.g. CIVL [48]) to formally verify that the group of data-race free microbenchmarks is indeed data race free.

4.1.1 *Helgrind.*

Helgrind [2] is a Valgrind-based error detecting 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 provide 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. We use Valgrind version 3.12.0 (built by GCC 4.9.3) and GCC version 4.9.3 to compile the benchmark suite.

4.1.2 *ThreadSanitizer.*

ThreadSanitizer (Tsan) [6] 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 on 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.

4.1.3 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 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 [5], 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 runtimes.

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

4.1.4 Intel Inspector.

Intel Inspector [3] is a dynamic analysis tool that detects threading and memory errors in C, C++ and Fortran codes. It supersedes Intel's Thread Checker tool [32, 38], 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.

4.2 Experiment Results

Our testing platform is the Quartz cluster hosted at the Livermore Computing Center [4]. Each computation node of the cluster has two Intel 18-core Xeon E5-2695 v4 processors with hyper threading support (18 cores × 2 sockets ×2 = 72 threads in total).

The first two columns in Table 5 list the IDs and names of all the microbenchmark programs. The file names have a format of $description - [orig|var] - yes|no.c$. The first part of a file name is some descriptive text for the program. The 2nd part with values of *orig* or *var* is optional. This is used to indicate the original version and var-length version of the same program. The 3rd part with values of *yes* or *no* indicates if this program has a known data race or not.

The evaluation is parameterized by three parameters: (i) the number of OpenMP threads in the execution, (ii) the data set size for the microbenchmarks in the var-length set, and (iii) the number of repeated runs of a microbenchmark with the exact same input. The environment variable *OMP_NUM_THREADS* is used to control the 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 72 microbenchmarks in total and 16 of them are in the var-length set. Each tool is run 5 times for every microbenchmark with the exact same input and independent of any other run. The experiment generates $(5 \times 7 \times 6 \times 16) + (5 \times 7 \times 1 \times 56) = 5320$ results and all are summarized in Table 5. Note that this indeed caused the tools to produce different results in some cases.

For a given race-no program, a chosen tool may report either false positive (FP) or true negative (TN) for different runs while true positive (TP) or false negative (FN) may be reported for a race-yes program.

In Table 5 the range of minimum and maximum number of races (min race - max race) is computed over all runs with a microbenchmark for each tool. For example, a min-max entry of 0-0 means that no data race was found in any run of the respective tool. An entry of 1-1 means in all runs exactly one data race was detected. If a range such as 3-187 is reported, this means that a different number of data races was detected and in every run at least 3 data races were detected. If a range starting at 0 is reported, for example `truedepsingleelement-orig-yes.c`, 0-255 (Archer), this means that sometimes no data race is detected, and sometimes one or more data races, up to 255, are detected. In this example, the microbenchmark is known to contain a data race ('Y' in column R). Therefore it is reported in the type-column that we get a true positive (TP, corresponding to more or equal than one data race) as well as a false-negative (FN, corresponding to zero data races detected) analysis result.

Table 6 summarizes the numbers of true/false positive and true/-false negative results for the four tools based on the results in Table 5. Again, for a same microbenchmark program with known data races, a dynamic analysis tool may generate either true positive (TP) or false negative results (FP) out of several runs. So there is a column named TP/FN in the table to indicate such cases. Similarly, there is a column named TN/FP for microbenchmark programs without data races.

The precision, recall and accuracy metrics of all the four tools are shown in Table 7. As discussed in Section 2, higher values are better for all the three metrics. We include a pair of numbers for each metric as the best and worst values calculated from the test results. The reason is that we conducted parameterized executions of the microbenchmark programs using a range of threads and

https://github.com/PRUNERS/archer 5ad2f47bc8ca8aad006a82a567179d2e0ce1ba75
https://github.com/OpenMPToolsInterface/LLVM-openmp.git
6e7140bf94d178f719200a6543558d7ae079183b

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
1	antidep1-orig-yes.c	Y	12 -	15	TP	3 -	188	TP	2 -	161	TP	1 -	1	TP
2	antidep1-var-yes.c	Y	11 -	26	TP	4 -	314	TP	3 -	371	TP	2 -	2	TP
3	antidep2-orig-yes.c	Y	15 -	18	TP	4 -	46	TP	4 -	36	TP	2 -	2	TP
4	antidep2-var-yes.c	Y	8 -	14	TP	3 -	188	TP	2 -	133	TP	1 -	1	TP
5	indirectaccess1-orig-yes.c	Y	13 -	15	TP	4 -	14	TP	1 -	1	TP	0 -	1	TP FN
6	indirectaccess2-orig-yes.c	Y	10 -	15	TP	3 -	14	TP	0 -	1	TP FN	0 -	1	TP FN
7	indirectaccess3-orig-yes.c	Y	11 -	15	TP	3 -	13	TP	0 -	1	TP FN	0 -	1	TP FN
8	indirectaccess4-orig-yes.c	Y	11 -	13	TP	3 -	14	TP	0 -	1	TP FN	0 -	1	TP FN
9	lastprivatemissing-orig-yes.c	Y	12 -	14	TP	3 -	161	TP	2 -	255	TP	1 -	1	TP
10	lastprivatemissing-var-yes.c	Y	11 -	14	TP	3 -	170	TP	2 -	255	TP	1 -	1	TP
11	minusminus-orig-yes.c	Y	12 -	16	TP	2 -	56	TP	2 -	49	TP	1 -	1	TP
12	minusminus-var-yes.c	Y	9 -	16	TP	2 -	128	TP	2 -	255	TP	1 -	1	TP
13	nowait-orig-yes.c	Y	15 -	19	TP	3 -	11	TP	0 -	1	TP FN	0 -	1	TP FN
14	outofbounds-orig-yes.c	Y	12 -	15	TP	3 -	87	TP	2 -	69	TP	1 -	1	TP
15	outofbounds-var-yes.c	Y	6 -	14	TP	3 -	221	TP	2 -	174	TP	1 -	1	TP
16	outputdep-orig-yes.c	Y	14 -	16	TP	3 -	109	TP	2 -	99	TP	2 -	2	TP
17	outputdep-var-yes.c	Y	11 -	16	TP	3 -	137	TP	2 -	255	TP	2 -	2	TP
18	plusplus-orig-yes.c	Y	14 -	17	TP	3 -	208	TP	3 -	355	TP	1 -	2	TP
19	plusplus-var-yes.c	Y	12 -	17	TP	4 -	267	TP	3 -	317	TP	1 -	2	TP
20	privatemissing-orig-yes.c	Y	12 -	15	TP	3 -	109	TP	2 -	99	TP	1 -	1	TP
21	privatemissing-var-yes.c	Y	8 -	14	TP	3 -	163	TP	2 -	255	TP	1 -	1	TP
22	reductionmissing-orig-yes.c	Y	13 -	16	TP	3 -	110	TP	2 -	99	TP	1 -	1	TP
23	reductionmissing-var-yes.c	Y	9 -	16	TP	3 -	161	TP	2 -	255	TP	1 -	1	TP
24	sections1-orig-yes.c	Y	10 -	12	TP	2 -	12	TP	1 -	1	TP	1 -	1	TP
25	simdtruedep-orig-yes.c	Y	0 -	0	FN	0 -	0	FN	0 -	0	FN	0 -	0	FN
26	simdtruedep-var-yes.c	Y	0 -	0	FN	0 -	0	FN	0 -	0	FN	0 -	0	FN
27	targetparallelfor-orig-yes.c	Y	24 -	30	TP	3 -	136	TP	2 -	121	TP	1 -	1	TP
28	taskdependmissing-orig-yes.c	Y	71 -	107	TP	2 -	12	TP	0 -	1	TP FN	0 -	1	TP FN
29	truedep1-orig-yes.c	Y	12 -	15	TP	3 -	80	TP	2 -	66	TP	1 -	1	TP
30	truedep1-var-yes.c	Y	8 -	14	TP	3 -	188	TP	2 -	162	TP	1 -	1	TP
31	truedepfirstdimension-orig-yes.c	Y	19 -	22	TP	6 -	307	TP	3 -	403	TP	0 -	2	TP FN
32	truedepfirstdimension-var-yes.c	Y	12 -	23	TP	4 -	325	TP	3 -	434	TP	0 -	2	TP FN
33	truedeplinear-orig-yes.c	Y	12 -	15	TP	3 -	100	TP	2 -	133	TP	1 -	1	TP
34	truedeplinear-var-yes.c	Y	8 -	14	TP	3 -	136	TP	2 -	97	TP	1 -	1	TP
35	truedepscalar-orig-yes.c	Y	14 -	17	TP	3 -	110	TP	2 -	99	TP	1 -	1	TP
36	truedepscalar-var-yes.c	Y	9 -	16	TP	3 -	222	TP	2 -	255	TP	1 -	1	TP
37	truedepseconddimension-orig-yes.c	Y	56 -	64	TP	5 -	194	TP	2 -	166	TP	1 -	1	TP
38	truedepseconddimension-var-yes.c	Y	55 -	67	TP	4 -	193	TP	2 -	167	TP	1 -	1	TP
39	truedepsingleelement-orig-yes.c	Y	12 -	14	TP	2 -	98	TP	0 -	255	TP FN	0 -	1	TP FN
40	truedepsingleelement-var-yes.c	Y	7 -	13	TP	2 -	168	TP	0 -	255	TP FN	0 -	1	TP FN
41	3mm-parallel-no.c	N	196 -	200	FP	0 -	532	FP TN	0 -	0	TN	0 -	0	TN
42	3mm-tile-no.c	N	129 -	141	FP	0 -	1	FP TN	0 -	0	TN	0 -	0	TN
43	adi-parallel-no.c	N	184 -	195	FP	2 -	255	FP	0 -	0	TN	0 -	0	TN
44	adi-tile-no.c	N	207 -	222	FP	14 -	295	FP	0 -	0	TN	0 -	0	TN
45	doall1-orig-no.c	N	6 -	11	FP	1 -	11	FP	0 -	0	TN	0 -	0	TN
46	doall2-orig-no.c	N	9 -	11	FP	1 -	11	FP	0 -	0	TN	0 -	0	TN
47	doallchar-orig-no.c	N	6 -	11	FP	1 -	11	FP	0 -	0	TN	0 -	0	TN
48	firstprivate-orig-no.c	N	7 -	11	FP	1 -	11	FP	0 -	0	TN	0 -	0	TN
49	fprintf-orig-no.c	N	9 -	11	FP	1 -	11	FP	0 -	0	TN	0 -	0	TN
50	functionparameter-orig-no.c	N	9 -	11	FP	1 -	11	FP	0 -	0	TN	0 -	0	TN
51	getthreadnum-orig-no.c	N	8 -	11	FP	1 -	11	FP	0 -	0	TN	0 -	0	TN
52	indirectaccesssharebase-orig-no.c	N	9 -	12	FP	3 -	13	FP	0 -	0	TN	0 -	0	TN
53	inneronly1-orig-no.c	N	48 -	55	FP	2 -	12	FP	0 -	0	TN	0 -	0	TN
54	inneronly2-orig-no.c	N	54 -	63	FP	4 -	108	FP	0 -	0	TN	0 -	0	TN
55	jacobi2d-parallel-no.c	N	104 -	116	FP	34 -	221	FP	0 -	0	TN	0 -	0	TN
56	jacobi2d-tile-no.c	N	79 -	90	FP	11 -	147	FP	0 -	0	TN	0 -	0	TN
57	jacobiinitialize-orig-no.c	N	6 -	11	FP	1 -	11	FP	0 -	0	TN	0 -	0	TN
58	jacobikernel-orig-no.c	N	86 -	94	FP	9 -	417	FP	0 -	1	FP TN	0 -	0	TN
59	lastprivate-orig-no.c	N	10 -	12	FP	2 -	12	FP	0 -	0	TN	0 -	0	TN
60	matrixmultiply-orig-no.c	N	5 -	11	FP	1 -	11	FP	0 -	0	TN	0 -	0	TN
61	matrixvector1-orig-no.c	N	9 -	11	FP	1 -	11	FP	0 -	0	TN	0 -	0	TN
62	matrixvector2-orig-no.c	N	51 -	60	FP	3 -	140	FP	0 -	1	FP TN	0 -	0	TN
63	outeronly1-orig-no.c	N	6 -	11	FP	1 -	11	FP	0 -	0	TN	0 -	0	TN
64	outeronly2-orig-no.c	N	6 -	11	FP	1 -	11	FP	0 -	0	TN	0 -	0	TN
65	pireduction-orig-no.c	N	5 -	11	FP	2 -	60	FP	0 -	1	FP TN	0 -	0	TN
66	pointernoaliasing-orig-no.c	N	6 -	11	FP	1 -	13	FP	0 -	0	TN	0 -	0	TN
67	restrictpointer1-orig-no.c	N	11 -	13	FP	1 -	13	FP	0 -	0	TN	0 -	0	TN
68	restrictpointer2-orig-no.c	N	9 -	14	FP	1 -	14	FP	0 -	0	TN	0 -	0	TN
69	sectionslock1-orig-no.c	N	9 -	12	FP	2 -	12	FP	0 -	0	TN	0 -	0	TN
70	simd1-orig-no.c	N	0 -	0	TN	0 -	0	TN	0 -	0	TN	0 -	0	TN
71	targetparallelfor-orig-no.c	N	22 -	28	FP	1 -	11	FP	0 -	0	TN	0 -	0	TN
72	taskdep1-orig-no.c	N	87 -	123	FP	2 -	13	FP	0 -	0	TN	0 -	1	FP TN

Table 5: Data race detection report (column R: whether a program contains a data race)

Tool	Race:Yes			Race:No		
	TP	TP/FN	FN	TN	TN/FP	FP
Helgrind	38	0	2	1	0	31
ThreadSanitizer	38	0	2	1	2	29
Archer	31	7	2	29	3	0
Intel Inspector	28	10	2	31	1	0

Table 6: Positive and negative results of the tools

array sizes, and the tools generate different results for the same program. For example, Archer generated both true positive and false negative for `indirectaccess2-orig-yes.c`. For tools where the results type column in Table 5 had two different results, we use the true-positive or true-negative one to calculate the best metric values while the false-positive and false-negative ones are used to calculate the worst metric values. From the results, it is clear that Archer and Intel Inspector, as OpenMP-aware tools, have higher precision than Helgrind and ThreadSanitizer do. The range of recall metrics for Intel Inspector is relatively large compared to other tools.

Tool	Precision		Recall		Accuracy	
	min	max	min	max	min	max
Helgrind	0.551	0.551	0.950	0.950	0.542	0.542
ThreadSanitizer	0.551	0.567	0.950	0.950	0.542	0.569
Archer	0.912	1.000	0.775	0.950	0.833	0.972
Intel Inspector	0.966	1.000	0.700	0.950	0.819	0.972

Table 7: Metrics for the tools

Details in tuning the analysis tools, observations and issues encountered in the evaluation process, are as follows:

- Intel Inspector configured with the narrow analysis scope is less accurate in detecting data races, but consumes much less overhead in analysis time. We chose the widest scope, customized the data race analysis byte granularity to one byte, used a stack frame depth of 16 and used maximum resources in the results shown in Table 5.
- Large data set for the microbenchmarks in var-length list will cause a segmentation fault in the tests with Archer. This issue can be avoided by increasing the stack size limitation (set to unlimited in our experiments).
- Compilers have different vectorization strategies in handling microbenchmarks with the OpenMP SIMD directives. We observed that GCC and Clang compilers would skip vectorization if a dependence is detected in the loop with an OpenMP SIMD directive (e.g. micro-benchmarks/simdtruedep-orig-yes.c) by reviewing the compiler optimization report. Their dependence analyses determine the vectorization for loops with OpenMP SIMD directive. For the Intel compiler, the -*qopenmp* option will enable *-qopenmp-simd* option and force vectorization regardless the result of dependence analysis.
- We have observed 12 runtime errors in the tests with Intel Inspector by reviewing the runtime log files. Ten of the errors are from six microbenchmarks with data races (microbenchmarks with ID numbers 2, 4, 15, 17, 32, and 36) and the other two are from two microbenchmarks without data races (microbenchmarks with ID numbers 44 and 63)[¶]. These are (rare) random runtime errors which may be caused by issues in the system, the Intel compiler or the tool. For every of those runs where an error was observed, no result was produced by the tool. Therefore we did not include test instances with errors in the results and only 5308 results for Intel Inspector are summarized in Table 5. For every of those cases where a runtime error occurred, another run of the tool on the same test program did produce a result.

5 RELATED WORK

Several benchmark suites have been written to test the performance of OpenMP systems. The NAS/OpenMP benchmark [22] and SPEComp [10] were designed to test parallel performance using highly optimized numeric codes. The OpenMP Source Code Repository (OpmSCR) [18] implements a common set of parallel programs in C, C++, and Fortran, and aims to be more representative of average OpenMP programming styles than either of the previous. The Rodinia [16] suite was written to test heterogeneous architectures, such as NUMA systems, FPGAs, and GPGPU computing. However, none of these collections are designed for data race detection. Any race found in one of these collections would be considered a bug and fixed, limiting its usefulness for verification.

Parallel benchmarks have been created for the Java language as well. When analyzing race-detectors many papers pull examples from pre-existing benchmark collections. BugBench [27] contains two test cases with known data races. Many publications have used examples from the Java Grande [43] benchmark suite that have races added. Finally, modified examples from the DaCapo [14] benchmark suite have been used to test race detector performance.

All of the tools evaluated in this paper are dynamic; they run the target program under instrumentation and analyze the execution trace [47]. 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 [25]. 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 [19, 20], expanding it to single-threaded event-driven programs [29], and defining additional relations such as casually-precedes [42].

Lockset analyses such as Eraser [40] 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 [21, 31, 35, 41, 47]. 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

Benchmark IDs listed in Table 5

enabling long-term [47] and large-scale [41] 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 [36] 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 [23] by performing pointer analysis in order to find global variables that are locally aliased. The RELAY analysis [45] modularizes each source of unsoundness in its analysis so that more accurate methods can be substituted when they are developed. OmpVerify [13] 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 [24], 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 [7, 46] analyzed several data race detection tools and compared the accuracy and performance of each. The first [7] 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 [46] 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.

6 CONCLUSION

In this paper, we presented DataRaceBench, a benchmark suite, for systematically evaluating data race detection tools on OpenMP programs. DataRaceBench includes microbenchmarks with and without data races. The benchmark suite is designed to meet a range of criteria related to scalability, generality, extensibility, and correctness.

We used DataRaceBench to evaluate four data race detection tools: Helgrind, ThreadSanitizer, Archer, and Intel Inspector. Our experiments show that DataRaceBench demonstrates the strengths and limitations of these tools. In particular, our work has the following findings:

(1) OpenMP awareness is required for a data race detection tool to reduce false positives (as also reported in [37]). Helgrind and ThreadSanitizer which are designed to support lower-level Pthreads but not OpenMP, report many false positive warnings. OpenMP programs have many synchronization points, often buried within the OpenMP runtime library calls without using Pthreads APIs. Therefore, a data race detection tool without knowledge of such OpenMP synchronization semantics can generate false positive warnings.

(2) SIMD instructions are overlooked by existing tools focusing on thread-level parallelism. An OpenMP compiler may refuse to vectorize loops annotated with SIMD directives when it detects dependencies within the loops. Even when such SIMD loops are forcefully vectorized, a dynamic analysis tool designed to capture thread level data races cannot detect instruction level data races.

(3) Helgrind, ThreadSanitizer, and Archer all reported multiple data race instances caused by the same pair of source code locations. Only Intel Inspector created user friendly reports that consolidated multiple data races found at the same source location to one event before reporting data races to users.

(4) Dynamic testing tools, likely will not find all data races during a single execution even in the presented microbenchmarks. In addition, they may not provide the same analysis results for the same test input and execution configurations. Therefore we recommend users of these tools to run them multiple times with varied inputs to increase the likelihood of observing data races.

(5) Dynamic testing tools can be sensitive to the number of threads used (and the runtime scheduling policies) when detecting data races in an OpenMP program. The reason is that a pair of loop iterations with loop-carried data dependence may be scheduled to the same thread, which may lead to false negative results. Therefore we recommend users of these tools to use a different number of threads and/or scheduling policies to increase the likelihood of observing data races.

(6) How a tool is configured can have a significant impact on the discovered data races. For example, Intel Inspector monitors memory accesses at 4-byte granularity by default to reduce runtime overhead. Using its default configuration, it failed to catch data races present in our test program operating on an array of characters (1-byte per character). We had to configure the Intel Inspector to use 1-byte granularity in order to generate expected results.

In the future, our choice of OpenMP will allow us to test tools on more highly threaded platforms, such as GPUs and Xeon Phi. We will use DataRaceBench to evaluate more tools, in particular tools using static analysis and formal verification techniques. In addition we may also add examples with benign data races to determine how tools handle them.

ACKNOWLEDGMENTS

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344, Lawrence Livermore National Security, LLC, via projects LDRD 17-ERD-023 and ProVESA KJ0401/KJ0402. IM release number LLNL-CONF-728717.

REFERENCES

[1] 1995. SPEC's Benchmarks. (1995). Retrieved July, 2017 from http://www.spec.org/benchmarks.html

[2] 2000. Helgrind. (2000). Retrieved July, 2017 from http://valgrind.org/docs/manual/hg-manual.html

[3] 2017. Intel Inspector 2017. (2017). Retrieved July, 2017 from https://software.intel.com/en-us/intel-inspector-xe

[4] 2017. Livermore Computing Quartz system. (2017). Retrieved July, 2017 from https://hpc.llnl.gov/hardware/platforms/Quartz

[5] 2017. LLVM Framework for High-Level Loop and Data-Locality Optimizations. (2017). Retrieved July, 2017 from https://polly.llvm.org

[6] 2017. ThreadSanitizer. (2017). Retrieved July, 2017 from https://github.com/google/sanitizers

[7] J. S. Alowibdi and L. Stenneth. 2013. An empirical study of data race detector tools. In *2013 25th Chinese Control and Decision Conference (CCDC)*. 3951–3955. https://doi.org/10.1109/CCDC.2013.6561640

[8] Cyrille Artho, Klaus Havelund, and Armin Biere. 2003. High-level data races. *Software Testing, Verification and Reliability* 13, 4 (2003), 207–227. https://doi.org/10.1002/stvr.281

[9] Vishal Aslot, Max Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B Jones, and Bodo Parady. 2001. SPEComp: A new benchmark suite for measuring parallel computer performance. In *International Workshop on OpenMP Applications and Tools*. Springer, 1–10.

[10] Vishal Aslot, Max J. Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B. Jones, and Bodo Parady. 2001. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In *OpenMP Shared Memory Parallel Programming, International Workshop on OpenMP Applications and Tools, WOMPAT 2001, West Lafayette, IN, USA, July 30-31, 2001 Proceedings (Lecture Notes in Computer Science)*, Rudolf Eigenmann and Michael Voss (Eds.), Vol. 2104. Springer, 1–10. https://doi.org/10.1007/3-540-44587-0_1

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

[12] Kunal Banerjee, Soumyadip Banerjee, and Santonu Sarkar. 2016. Data-race detection: the missing piece for an end-to-end semantic equivalence checker for parallelizing transformations of array-intensive programs. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ACM, 1–8.

[13] V. Basupalli, Tomofumi Yuki, Sanjay V. Rajopadhye, Antoine Morvan, Steven Derrien, Patrice Quinton, and David Wonnacott. 2011. ompVerify: Polyhedral Analysis for the OpenMP Programmer. In *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)*, Barbara M. Chapman, William D. Gropp, Kalyan Kumaran, and Matthias S. Müller (Eds.), Vol. 6665. Springer, 37–53. https://doi.org/10.1007/978-3-642-21487-5_4

[14] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. ACM Press, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[15] Prasanth Chatarasi, Jun Shirako, and Vivek Sarkar. 2016. Static Data Race Detection for SPMD Programs via an Extended Polyhedral Representation. (2016).

[16] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Ieee, 44–54.

[17] Jack J Dongarra, Piotr Luszczek, and Antoine Petitet. 2003. The LINPACK benchmark: past, present and future. *Concurrency and Computation: practice and experience* 15, 9 (2003), 803–820.

[18] Antonio J Dorta, Casiano Rodriguez, and Francisco de Sande. 2005. The OpenMP source code repository. In *Parallel, Distributed and Network-Based Processing, 2005. PDP 2005. 13th Euromicro Conference on*. IEEE, 244–250.

[19] Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-Juergen Boehm. 2012. IFRit: interference-free regions for dynamic data-race detection. In *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*, Gary T. Leavens and Matthew B. Dwyer (Eds.). ACM, 467–484. https://doi.org/10.1145/2384616.2384650

[20] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 121–133. https://doi.org/10.1145/1542476.1542490

[21] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal sound predictive race detection with control flow abstraction. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 337–348. https://doi.org/10.1145/2594291.2594315

[22] Hao-Qiang Jin, Michael Frumkin, and Jerry Yan. 1999. The OpenMP implementation of NAS parallel benchmarks and its performance. (1999).

[23] Vineet Kahlon, Yu Yang, Sriram Sankaranarayanan, and Aarti Gupta. 2007. Fast and Accurate Static Data-Race Detection for Concurrent Programs. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings (Lecture Notes in Computer Science)*, Werner Damm and Holger Hermanns (Eds.), Vol. 4590. Springer, 226–239. https://doi.org/10.1007/978-3-540-73368-3_26

[24] Young-Joo Kim, Daeyoung Kim, and Yong-Kee Jun. 2008. An Empirical Analysis of Intel Thread Checker for Detecting Races in OpenMP Programs. In *7th IEEE/ACIS International Conference on Computer and Information Science, IEEE/ACIS ICIS 2008, 14-16 May 2008, Portland, Oregon, USA*, Roger Y. Lee (Ed.). IEEE Computer Society, 409–414. https://doi.org/10.1109/ICIS.2008.79

[25] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565. https://doi.org/10.1145/359545.359563

[26] Chunhua Liao, Daniel J Quinlan, Jeremiah J Willcock, and Thomas Panas. 2010. Semantic-aware automatic parallelization of modern applications using high-level abstractions. *International Journal of Parallel Programming* 38, 5 (2010), 361–378.

[27] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. 2005. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the evaluation of software defect detection tools*, Vol. 5.

[28] Hongyi Ma, Steve R Diersen, Liqiang Wang, Chunhua Liao, Daniel Quinlan, and Zijiang Yang. 2013. Symbolic analysis of concurrency errors in openmp programs. In *Parallel Processing (ICPP), 2013 42nd International Conference on*. IEEE, 510–516.

[29] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. 2014. Race detection for Android applications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 316–325. https://doi.org/10.1145/2594291.2594311

[30] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. 2007. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 22–31. https://doi.org/10.1145/1250734.1250738

[31] Robert O'Callahan and Jong-Deok Choi. 2003. Hybrid dynamic data race detection. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2003, June 11-13, 2003, San Diego, CA, USA*, Rudolf Eigenmann and Martin C. Rinard (Eds.). ACM, 167–178. https://doi.org/10.1145/781498.781528

[32] Paul Petersen and Sanjiv Shah. 2003. OpenMP support in the Intel® thread checker. In *International Workshop on OpenMP Applications and Tools*. Springer, 1–12.

[33] Louis-Noël Pouchet. 2012. PolyOpt/C:a Polyhedral Optimizer for the ROSE compiler. (2012). Retrieved July, 2017 from http://web.cs.ucla.edu/~pouchet/software/polyopt/

[34] Louis-Noël Pouchet and Tomofumi Yuki. 2011. PolyBench/C. (2011). https://sourceforge.net/projects/polybench/

[35] Eli Poznianski and Assaf Schuster. 2003. 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*. IEEE Computer Society, 287. https://doi.org/10.1109/IPDPS.2003.1213513

[36] Polyvios Pratikakis, Jeffrey S. Foster, and Michael W. Hicks. 2006. LOCKSMITH: context-sensitive correlation analysis for race detection. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, Michael I. Schwartzbach and Thomas Ball (Eds.). ACM, 320–331. https://doi.org/10.1145/1133981.1134019

[37] J. Protze, S. Atzeni, D. H. Ahn, M. Schulz, G. Gopalakrishnan, M. S. MÃijller, I. Laguna, Z. Rakamaric, and G. L. Lee. 2014. Towards Providing Low-Overhead Data Race Detection for Large OpenMP Applications. In *2014 LLVM Compiler Infrastructure in HPC*. 40–47. https://doi.org/10.1109/LLVM-HPC.2014.7

[38] Paul Sack, Brian E Bliss, Zhiqiang Ma, Paul Petersen, and Josep Torrellas. 2006. 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*. ACM, 34–41.

[39] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comput. Syst.* 15, 4 (Nov. 1997), 391–411. https://doi.org/10.1145/265924.265927

[40] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comput. Syst.* 15, 4 (1997), 391–411. https://doi.org/10.

1145/265924.265927

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

[42] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. Sound predictive race detection in polynomial time. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 387–400. https://doi.org/10.1145/2103656.2103702

[43] L. A. Smith, J. M. Bull, and J. Obdrizalek. 2001. A Parallel Java Grande Benchmark Suite. In *Supercomputing, ACM/IEEE 2001 Conference*. 6–6. https://doi.org/10.1145/582034.582042

[44] Michael Süß and Claudia Leopold. 2008. Common mistakes in OpenMP and how to avoid them. In *OpenMP Shared Memory Parallel Programming*. Springer, 312–323.

[45] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 2007. RELAY: static race detection on millions of lines of code. In *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*, Ivica Crnkovic and Antonia Bertolino (Eds.). ACM, 205–214. https://doi.org/10.1145/1287624.1287654

[46] Misun Yu, Seung-Min Park, Ingeol Chun, and Doo-Hwan Bae. 2017. Experimental Performance Comparison of Dynamic Data Race Detection Techniques. *ETRI Journal* 39, 1 (02 2017), 124–134. https://doi.org/10.4218/etrij.17.0115.1027

[47] Yuan Yu, Tom Rodeheffer, and Wei Chen. 2005. RaceTrack: efficient detection of data race conditions via adaptive tracking. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005*, Andrew Herbert and Kenneth P. Birman (Eds.). ACM, 221–234. https://doi.org/10.1145/1095810.1095832

[48] Manchun Zheng, Michael S. Rogers, Ziqing Luo, Matthew B. Dwyer, and Stephen F. Siegel. 2015. CIVL: Formal Verification of Parallel Programs. In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, Myra B. Cohen, Lars Grunske, and Michael Whalen (Eds.). IEEE Computer Society, 830–835. https://doi.org/10.1109/ASE.2015.99

A ARTIFACT DESCRIPTION

A.1 Overview

A.1.1 How software can be obtained (if available). DataRaceBench can be downloaded from https://github.com/LLNL/dataracebench .

A.1.2 Hardware dependencies. The computation node we used is from the Quartz Intel cluster as described at https://computing.llnl.gov/tutorials/lc_resources/#IntelSystems . The nodes there are machines with Intel Xeon multicore processors, running Linux operating systems. Any similar hardware supporting execution of OpenMP programs should be supported.

A.1.3 Software dependencies. As a benchmark suite to evaluate data race detection tools for OpenMP programs, you need to install the tools being evaluated and a corresponding compilers supporting OpenMP. Bash is also needed if you want to use the execution bash script to automate your experiments.

The four dynamic analysis tools (Helgrind, ThreadSanitizer, Archer, Intel inspector) needed to reproduce our results are:

- Helgrind is part of Valgrind 3.12.0
- ThreadSanitizer is the one based on Clang/LLVM 4.0.1
  (https://github.com/llvm-mirror/clang.git 559aa046fe3260d8640791f2249d7b0d458b5700)
  (https://github.com/llvm-mirror/llvm.git 08142cb734b8d2cefec8b1629f6bb170b3f94610)
- Archer is from the towards_tr4 branch with OpenMP runtime support in the towards_tr4 branch from OMPT. The tool is built using Clang/LLVM 4.0.1
  (https://github.com/PRUNERS/archer/tree/towards_tr4 5ad2f47bc8ca8aad006a82a567179d2e0ce1ba75)
  (https://github.com/OpenMPToolsInterface/LLVM-openmp.git 6e7140bf94d178f719200a6543558d7ae079183b)
- Intel Inspector 2017 (build 475470) requires Intel compiler version 17.0.2

A.1.4 Datasets. All programs in DataRaceBench have builtin data sets. No additional input files are needed. A command line option is used to specify different sizes of arrays in some programs.

A.2 Installation

- You need to follow the installation instructions of the race detection tools you want to evaluate.
- DataRaceBench is run with the provided script.

A.3 Evaluation Workflow

There are three major steps to use DataRaceBench:

1. The first step is to install and setup your data race detection tools you want to evaluate.
2. The second step is to configure the execution script in DataRaceBench.
3. The final step is to run the script and generate evaluation results as csv files.

DataRaceBench's execution script, check-data-races.sh, has builtin support for Helgrind, ThreadSanitizer, Archer, and Intel Inspector (exploiting maximum resources). You don't have to modify the script unless you want to try a different tool. Finally, you run the script to evaluate one or more tools you are interested in. The full DataRaceBench test suite is run using `check-data-races.sh` by selecting one of the four supported tools by listing its name (helgrind, tsan, archer, inspector).

The script also allows to run all four tools at once with small settings (option `--small`) or perform a trial run to verify that every microbenchmark can be compiled and executed (option `--run`).

```
# show more help information for this script
./check-data-races.sh --help

Usage: ./check-data-races.sh [--run] [--help]

--help     : this option
--small    : compile and test all benchmarks using small parameters with
              Helgrind, ThreadSanitizer, Archer, Intel inspector.
--run      : compile and run all benchmarks with gcc (no evaluation)
--helgrind : compile and test all benchmarks with Helgrind
--tsan     : compile and test all benchmarks with clang ThreadSanitizer
--archer   : compile and test all benchmarks with Archer
--inspector: compile and test all benchmarks with Intel Inspector
```

A.4 Evaluation and Results

Running check-data-races.sh generates csv files stored in a sub-directory, named `results`, containing multiple lines of information. Each line indicates results of one or several experiments of a given tool under a certain configuration, with fields for:

- The name of the evaluated tool.
- The filename of the microbenchmark.
- True or false (indicating whether the microbenchmark is known to have a data race).
- The number of threads being used for execution.
- Varying length array size (reports N/A if the microbenchmark has no variable length array(s)).
- How many data races the tool reports for this experiment.
- The elapsed time reported in seconds in the experiment.

A.5 Experiment customization

The full DataRaceBench test suite is run using `check-data-races.sh` and calls the test harness script (`scripts/test-harness.sh`) to perform the actual tests. This test harness script can also be called directly and allows you to specify a number of additional parameters for the four tools (Valgrind, ThreadSanitizer, Archer, and Intel Inspector). The respective tool can be selected with the option `-x` followed by its name (`-x helgrind`, `-x tsan`, `-x archer`, `-x inspector`). For Intel Inspector two configuration variants are available: the default (`-x inspector`) configuration and the configuration using maximum resources (`-x inspector-max-resources`). Furthermore the following options can be set:

- How many times a microbenchmark should be run (option `-n` followed by the respective number).
- The number of threads used in a microbenchmark (option `-t` followed by the respective number).
- The size of the variable length array (only relevant for varlen microbenchmarks) (option `-d` followed by the respective number).

An overview of these options can be obtained by invoking the script with `--help`. Note that the script also automatically selects the required compiler for the respective tool.