LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-825647

High-Precision Evaluation of Both Static andDynamic Tools using DataRaceBench

P. Lin, C. Liao

August 10, 2021

Correctness 2021: Fifth International Workshop on Software Correctness for HPC Applications
St. Louis, MO, United States
November 19, 2021 through November 19, 2021

Disclaimer

High-Precision Evaluation of Both Static and Dynamic Tools using DataRaceBench

Pei-Hung Lin
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory, CA, USA
Livermore, California, USA
lin32@llnl.gov

Chunhua Liao
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, California, USA
liao6@llnl.gov

***Abstract*—DataRaceBench (DRB) is a dedicated benchmark suite to evaluate tools aimed to find data race bugs in OpenMP programs. Using microbenchmarks with or without data races, DRB is able to generate standard quality metrics and provide systematical and quantitative assessments of data race detection tools. In this paper, we present a new version of DRB with several improvements. First, we design a novel approach to enable high-precision checking of tool results. The approach relies on a format to accurately encode data race ground truth including variables, read/write types, and source file location information. The test harness of DRB has also been improved to support static data race detection tools. Finally, an enhanced code similarity analysis is developed to consider code region details and cover more regions. Our experiments show that the improved DRB generates more accurate reports and exposes more limitations of both static and dynamic data race detection tools. The enhanced similarity analysis also is able to guide us to investigate similar code regions in DataRaceBench in more detail.**

***Index Terms*—Benchmarks, OpenMP, Data Races, Tools**

I. Introduction

DataRaceBench (DRB) is a dedicated benchmark suite to evaluate tools aimed to find data race bugs in OpenMP programs. Since its initial release in 2017, DRB has incorporated various additions to have a richer set of microbenchmark programs to cover the latest OpenMP constructs, base programming languages (such as C/C++ and Fortran) and modern parallel hardware devices (e.g. GPUs). Using microbenchmarks with or without data races, DRB is able to generate standard quality metrics (such as accuracy and F-1 score) and provide systematical and quantitative assessments of data race detection tools. The existing workflow using DRB has several steps: compiling the microbenchmark source codes, running tools being evaluated, collecting the reports generated by the tools, and processing the report files against ground truth to calculate quality metrics of a tool. Due to its good design and automated workflow, DRB has been widely adopted by tool developers [1]–[7].

However, DRB still has several limitations as reported by users. A major limitation is its last step of the workflow, which uses a simple file-level true/false evaluation. If a tool reports a data race for a microbenchmark file with some known data races, DRB simply counts it as a true positive. This works fine since most likely, a microbenchmark file with known data races has only a single pair of data race locations. However, a tool could report a wrong pair of data race locations but DRB still would count it as a correct data race (true positive) report. This inadequacy in the workflow may lead to incorrect evaluation results.

Another limitation is that the test harness for DRB was designed mainly for the dynamic tools that require application execution as part of the race detection process. With more static analysis tools becoming available, the existing test framework for DRB will not be adequate and requires revision to support the needs.

A third limitation is that the similarity analysis in prior work [8] is primitive. It only considers properties of OpenMP directives associated with each code region, leaving out features inherent to the code region itself. As we try to add more microbenchmarks into DRB, it becomes increasingly important to have a good quality of similarity analysis to avoid duplicated microbenchmarks being collected in DRB. The analysis helps us have a minimum collection of microbenchmark programs with maximum coverage of OpenMP code patterns with or without data races.

In this paper, we present the enhancements added into the DRB to address the limitations mentioned above. The following lists the contributions introduced into DRB:

1) We design a format to accurately encode the ground truth of data races in DRB microbenchmarks. A high-precision evaluation method is designed based on information parsed from the ground truth and tool reports.
2) We add support for static data race detection tools in order to generate a complete dashboard reporting the state-of-the-art of data race detection.
3) An enhanced code similarity analysing is developed to detect duplicate test cases within DRB.

The remainder of this paper is organized as follows. In the next section, we introduce a method to enable high-precision, fine-grain correctness checking using DRB. Section III presents how static data race detection tools are supported. Section IV describes an enhanced similarity analysis. Experiments are discussed in Section V. Finally, related work

Prepared by LLNL under Contract DE-AC52-07NA27344 (LLNL-CONF-825647).

is mentioned in Section VI and conclusions are drawn in Section VII.

II. High-Precision Correctness Checking

DRB includes both race-yes and race-no microbenchmarks. The race-yes microbenchmarks have known data races injected purposely to evaluate the data race detection capability of a given tool. The race-no microbenchmarks do not have any data races. Whenever possible, a race-yes microbenchmark is put into a single source file containing only a single pair of source locations that cause data races.

This design simplifies the validation of tool generated reports. For a race-no microbenchmark, if a tool reports any data races, it is counted as a false positive. Otherwise, a true negative is counted. Similarly for a race-yes microbenchmark, if a tool reports any data races, it is counted as a true positive. Otherwise, a false negative is counted. The test harness script of DRB does not check details of tool reports such as variable names and read/write properties. Essentially, this is a coarse-grain, file-level validation of tool results.

Over time, microbenchmarks with more pairs of data race locations have been added. It is possible that a tool reports wrong pairs of source locations causing data races for a given source file. But the test harness of DRB still counts the tool with a true positive. To address this limitation, we design a new high-precision, fine-grain correctness checking method based on accurate encoding of ground truth, formalization of tool reports, and enhanced correctness evaluation techniques. This section discusses the details of this method.

A. Encoding Ground Truth

For each known data race in a source file, we must represent the following key information: 1) the shared variables causing the race, 2) the pair of source locations including the line and column information of the variable accesses, and 3) the read/write type of each variable access. They are considered the ground truth of the existence of data races of a source file.

We propose to encode the ground truth information as part of the comment to a program. Syntax listed in Listing 1 is used to present a source location information of a variable access. `VAR_NAME` includes the variable name and subscript information if the access is an array access. Three fields separated by the colon symbol are the `LINE_INFO` representing the line number of the variable access in the source code, the `COLUMN_INFO` showing the column number of the first character of the variable name, and letter `R` or `W` identifying the read and write access respectively. A pair of source location information separated by `vs.` represent the pair of variable accesses that causes a data race in the given source code (Listing 2). Note that each variable access encoding should have only one R or W property. If a variable is both Read and Written, the accesses should be encoded separately in different data race pairs.

Listing 1: Encoding syntax of the variable access information

```
{VAR_NAME}@{LINE_INFO}:{COLUMN_INFO}:{R/W}
```

Listing 2: Encoding syntax of a pair of variable accesses

```
{VAR_NAME_1}@{LINE_INFO_1}:{COLUMN_INFO_1}:{R/W} vs.
{VAR_NAME_2}@{LINE_INFO_2}:{COLUMN_INFO_2}:{R/W}
```

Listing 3 shows the DRB001 microbenchmark that has a loop carried anti-dependence causing a data race, using the designed encoding method. The order of these two variable access information is not critical. It will be normalized by an evaluation script.

Listing 3: C: DRB001-antidep1-orig-yes.c

```
/*
A loop with loop-carried anti-dependence.
Data race pair: a[i+1]@64:10:R vs. a[i]@64:5:W
*/
#include <stdio.h>
int main(int argc, char* argv[])
{
  int i;
  int len = 1000;

  int a[1000];

  for (i=0; i<len; i++)
    a[i]= i;

#pragma omp parallel for
  for (i=0;i< len -1 ;i++)
    a[i]=a[i+1]+1;

  printf ("a[500]=%d\n", a[500] );
  return 0;
}
}
```

A second ground truth encoding syntax (shown in Listing 4) is needed to present multiple source location pairs generated from a read variable set and a write variable set. This syntax is concise by avoiding explicit listing of all pairs.

Listing 4: Encoding syntax for Read/Write set of variable accesses

```
Write_set = {W_NAME_1@W_LINE_1:W_COLUMN_1,
    W_NAME_2@W_LINE_2:W_COLUMN_2...}
Read_set  = {R_NAME_1@R_LINE_1:R_COLUMN_1,
    R_NAME_2@R_LINE_2:R_COLUMN_2...}
```

DRB095, shown in Listing 5, is an example using the read/write sets to encode multiple pairs of data races. Any combination of two elements from these two different sets can cause a read/write data race for DRB095. Similarly, any combination of two elements from the write variable set can cause a write/write data race. Note that the `j@69:30` from the `j++` expression for the loop iteration expression represents both read and write variable accesses due to the usage of the `++` operator.

A Python script, `getSourceRaceInfo.py`, is provided to parse the ground truth information written in the source codes and save relevant information into files in JSON format. Therefore, each microbenchmark will have an associated JSON file storing the data race ground truth information.

B. Parsing Different Data Race Tool Reports

Data race detection tools often generate customized reports using different formats presenting different information regarding data races found. Table I lists five existing tools that

Listing 5: C: DRB095-doall2-taskloop-orig-yes.c

```
/*
Two-dimensional array computation:
Only one loop is associated with omp taskloop.
The inner loop's loop iteration variable will be
    shared if it is shared in the enclosing context.
Data race pairs (we allow multiple ones to preserve
    the pattern):
  Write_set = {j@69:14, j@69:30, a[i][j]@70:11}
  Read_set = {j@69:21, j@69:30, j@70:16, a[i][j]@70
    :11}
  Any pair from Write_set vs. Write_set  and
    Write_set vs. Read_set is a data race pair.
*/
#if (_OPENMP<201511)
#error "An OpenMP 4.5 compiler is needed to compile
    this test."
#endif
#include <stdio.h>
int a[100][100];
int main()
{
  int i, j;
#pragma omp parallel
  {
#pragma omp single
    {
#pragma omp taskloop
      for (i = 0; i < 100; i++)
        for (j = 0; j < 100; j++)
          a[i][j]+=1;
    }
  }
  printf ("a[50][50]=%d\n", a[50][50]);
  return 0;
}
```

are actively maintained and the available information in their reports with respect to the ground-truth including memory address, Read/Write type, line and column numbers. Some tools may also report inaccurate information for the column numbers. With various formats from various tools, formalizing the reports from various tools becomes a necessity to simplify the tool evaluation process.

Tool	Mem. Address	R/W type	Line	Column
Intel Inspector		✓	✓	
ROMP	✓		✓	incorrect for all
ThreadSanitizer	✓	✓	✓	
Coderrect			✓	incorrect for Fortran
LLOV		✓	✓	

TABLE I: Information available in tool reports.

For example, Intel Inspector by default reports only issue statistics including the number of races detected by the tool. It requires additional steps by a separated tool to retrieve source information and variable access type. However, the source column information is not provided in the report (shown in Listing 6).

ROMP by default reports the total number of detected races. ROMP reports a higher number of detected races because it reports races according to memory access in byte granularity. A race caused by a 4-byte variable access will lead to 4 separated reported races. An environment variable, "ROMP_REPORT_LINE=true", needs to be set up to enable the report for source information. However, the variable access type information is not available from the report.

Listing 6: Intel Inspector report format

```
P1: Error: Data race: New
 P1.9: Error: Data race: New
  DRB001-antidep1-orig-yes.c(64): Error X17: Write:
    Function main: Module DRB001-antidep1-orig-yes.c
    .inspector.out
  DRB001-antidep1-orig-yes.c(64): Error X18: Read:
    Function main: Module exec/DRB001-antidep1-orig-
    yes.c.inspector.out
```

Listing 7: ROMP report format

```
3717 CoreUtil.cpp:76] RAW: data race found at mem
    addr:       7fffe0d6e754
 DRB001-antidep1-orig-yes.c@[40083a]line:64 col:0 vs
 DRB001-antidep1-orig-yes.c@[400827]line:64 col:0
```

ThreadSanitizer reports all the data race instances triggered by different pairs of hardware threads, as shown in Listing 8. The total number of reported races varies according to the number of OpenMP threads used in the testing. ThreadSanitizer can report the source information (variable name and source line) and the read/write type information for the variable accesses causing the race.

Listing 8: ThreadSanitizer report format

```
WARNING: ThreadSanitizer: data race (pid=43)
  Read of size 4 at 0x7ffcfdbbdbb8 by thread T5:
    #0 .omp_outlined._debug__ DRB001-antidep1-orig-
    yes.c:64 (DRB001-antidep1-orig-yes.c.tsan-clang.
    out+0x4cc22e)
    ...
  Previous write of size 4 at 0x7ffcfdbbdbb8 by
    thread T6:
    #0 .omp_outlined._debug__ /RB001-antidep1-orig-
    yes.c:64 (DRB001-antidep1-orig-yes.c.tsan-clang.
    out+0x4cc25c)
    ...
```

Coderrect prints out verbose reports to identify the location of the detected race. Code snippet with annotation is listed to help users to find the variable access location in source code. The variable access type is not reported explicitly but can be manually retrieved from the listed code snippet.

LLOV reports the file path and source line information for the variable accesses causing a race. The variable access type information can also be found from the reported information.

For each tool's output format, a parser is developed to retrieve information from the tool report and store the information into JSON format.

C. High-Precision Correctness Evaluation

Using JSON files storing both the ground truth and tool reports, we have developed a high-precision correctness checking process with the following steps:

- Normalizing Data: This step filters out memory addresses which do not fit any source code locations. They may come from the OpenMP runtime library. All data race

pairs are also sorted using first the variable access type, the source code line information, and finally the source code column information.
- Generating Distinct Set of Races: A tool may report duplicated information. This step keeps only the distinct set of races for the final comparisons.
- Trivial Decision: A trivial decision can be made for three scenarios. First, a true negative can be recorded if there is no race recorded in either the ground truth or a tool report. A false positive can be recorded if there is no race in ground truth but a tool reports one. Finally, a false negative is counted if there is data race in ground truth but the tool does not find any.
- Fine-granularity Comparison: This step is used when both the ground truth and a tool report contain data races. The accuracy of evaluation can be adjusted by considering different combinations of variable access type, read/write type, as well as source line and source column information. DRB provides scripts to allow different levels of fine-granularity comparisons using different encoding information involved. For example, the first level will only consider source line/column information when deciding if a tool reports a correct data race. A more rigid level will additionally consider the read/write types reported. Evaluation is expected to be more precise when more information is considered in comparison.

The evaluation process performs comparison between the list of variable access pairs from the source encoded information and the list of variable access pairs reported by the tools. One detail is that while our ground truth explicitly encodes all true positive pairs, it does not explicitly encode the complete set of true negative pairs. The true negative pairs include all possible pairs of variable access locations which do not cause data races. Instead, we use an approximation set of true negative pairs in our correctness evaluation. The idea is that for each tool, the set of data race pairs reported can be categorized into two subsets: true positive subset and false positive subset. We can compare a tool's output against the True Positive ground truth and easily obtain its false positive set (which in turn is a subset of the ground truth true negative set). For all selected tools, we union their false positive sets as an approximation of the total true negative ground truth.

III. Static analysis tool support

State-of-the-art data race detection analyses can be largely divided into dynamic and static approaches. Majority of data race detection tool developments, for both academic and industrial usage, exploit the dynamic approaches, including Archer, ThreadSanitizer, Intel Inspector, and Helgrind. The interests for static data race detection tools are also growing as they complement dynamic approaches. DRACO [9], OMPRacer [10], and LLOV [11] are the three latest developed tools that were evaluated with DRB.

The testing harness of DRB was designed solely for the dynamic data race tools since only the dynamic tools were available several years ago. With more static tools becoming available, supporting static tool evaluation becomes a critical need in the development of DRB. A candidate static tool to be included into DRB should be publicly available, easy for use and has detailed instructions or container support. And most important of all, the static tool should support both C and Fortran languages and process the microbenchmarks collected in DRB without any modification. Coderrect [12], previously known as OMPRacer, and LLOV [11] fulfill the selection criteria to be included into DRB evaluation support.

Coderrect is developed as a Clang/LLVM extension for static data race analysis. The Fortran language support in Coderrect depends on the LLVM Fortran front-end development. Coderrect exploits multiple analyses to extract OpenMP semantics to build the static happens-before graph at the level of logical threads, reason OpenMP-defined data sharing semantic, and perform OpenMP semantic analysis with a set of novel algorithms for pointer aliases and value flows in an inter-procedural setting. LLOV is a lightweight, language agnostic, and static data race detection tool built on top of the LLVM framework. LLOV exploits Polly, the polyhedral compilation engine in LLVM, to perform exact dependence analysis to detect data races in affine regions. For source code in a non-affine region, LLOV checks the barriers or locks and uses Mod/Ref information from the LLVM's Alias Analysis to detect potential data races.

Both Coderrect and LLOV have been evaluated using DRB by their developers. We add support in the test harness script in DRB to include the environment setup, compiler command, evaluation command and the tool reporting commands for these two static tools. The required information to run those tools are available in the tool's source code repositories.

IV. Enhanced Similarity Analysis

DRB is expected to continue to grow and collect more microbenchmarks representing OpenMP data race code patterns extracted from different domains. Ideally, DRB should contain a minimum collection, if possible, to represent the maximum coverage in terms of code patterns and the OpenMP language specifications. Similarity analysis can help identify duplicated code patterns and guide the addition of new microbenchmarks.

A distance-based similarity analysis has been applied to DRB [8]. The analysis relies on the feature vectors representing both static and dynamic information of code regions. The static information includes OpenMP directives and clauses used in source code while dynamic information contains the data race analysis result. A major limitation of this analysis is that the feature vector misses information representing the content of the code region wrapped by OpenMP directives.

This paper overcomes the limitation by expanding the feature vectors to contain more fields representing the details of the code regions, as shown in Table II. Fields E123 through E134 are the new features added, including various statistics of a code region about its operators, integer and floating point constants, and variable references. The operators include Add, subtract, multiply, divide and assign operators. Comparison operators, bitwise operators, logical operators, and combined

assign operators (+=, -=, *=, etc.) are also counted. We revised a Clang-based plugin tool, the OpenMP Extractor [8], to collect these features from clang AST. The new feature vector is defined using the following formula:

$$\vec{A} = (Directive, Clause, \{Extracted\ source\ info.\ list\}, \{Tool\ result\ list\}, Ground\ Truth) \quad (1)$$

Another improvement is the coverage of code regions. In previous work the feature vectors were generated only for OpenMP parallel loops under the OMPLoopDirective AST node. We have expanded the coverage to include basic blocks under the OMPParallelDirective AST node. This will include more code regions into analysis. For example the `DRB074-flush-orig-yes.c`, that has only `omp parallel reduction` in the source code, was not previously considered since it does not have a loop. The enhanced similarity analysis can now cover it as part of a parallel region.

Feature	Value Range	Encoding Fields	Description
Directive	[0,1]	[E0-E86]	87 directives are flattened into the first 87 elements in the vector. The existing directive's value is set to 1, else 0.
Clause	[0,1]	[E87-E122]	The next 36 integer elements are 36 flattened clauses. If the value is 1, the test case contains that clause.
AddOp	[0,n]	E123	Number of Add operators.
SubOp	[0,n]	E124	Number of Subtract operators.
MulOp	[0,n]	E125	Number of Multiply operators.
DivOp	[0,n]	E126	Number of divide operators.
CompOp	[0,n]	E127	Number of compare operators.
BitOp	[0,n]	E128	Number of bit operators.
LogicOp	[0,n]	E129	Number of logic operators.
AssignOp	[0,n]	E130	Number of assign operators.
CombOp	[0,n]	E131	Number of combined operators (+=, -=, etc.).
ConstOp	[0,n]	E132	Number of constant integer and floating values.
VariableRef	[0,n]	E133	Number of distinct variable names.
totalVarRef	[0,n]	E134	Total number of variable references.
Intel	[-2,1]	E135	Data race result by the tool. -2 represents time out. -1 represents the segmentation fault. 0 represents no data race. 1 represents the data race.
ROMP	[-2,1]	E136	
Tsan	[-2,1]	E137	
Coderrect	[-2,1]	E138	
LLOV	[-2,1]	E139	
Ground Truth	[0,1]	E140	Ground Truth, whether a loop has a data race or not.

TABLE II: Feature vector's definition and encoding methods

To compare the similarity, this work continues to use Cosine Distance, which is a way to calculate the distance between two non-zero vectors of an inner-product space and is more suitable for high-dimensional vectors. The Cosine Distance is derived from the Euclidean dot product formula as follows:

$$\cos(\theta) = \frac{\vec{A} \cdot \vec{B}}{|\vec{A}||\vec{B}|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}} \quad (2)$$

The distance, range from [-1,1], is used to determine the similarity of two vectors. If two vectors are more similar, the cosine distance will be closer to 1, and the degree for two vectors will be closer to 0°.

V. Experiments

In this section, we present experiment results using the proposed high-precision correctness checking method to evaluate both static and dynamic tools. The selected tools are capable of detecting data races in OpenMP codes, publicly available and with active development and maintenance in the past year. We also evaluate the enhanced similarity analysis for its effectiveness.

A. Experiment Configuration

Table III summarizes both dynamic and static tools used in this evaluation. Intel Inspector was evaluated on the Quartz machine hosted at the Livermore Computing Center. Experiments with the rest tools were conducted using Docker containers. The containers for LLOV and ROMP are built from the published Docker images. The rest containers were built from the Dockerfiles provided in the DRB repository.

Tool	Tool type	Version	Compiler
Intel Inspector	Dynamic	2020(build 603904)	Intel Compiler 19.1.0.166
ROMP	Dynamic	20ac93c	GCC & gfortran 7.4
ThreadSanitizer	Dynamic	12.0	Clang/LLVM 12.0.0, gfortran 10.3.0
Coderrect	Static	0.8.0	Clang/LLVM 9.0
LLOV	Static	N/A	Clang/LLVM 6.0.1

TABLE III: Selected Tools: version and compilers used.

B. Evaluation results

The new version of DRB continues to automatically calculate counts of False Positive, True Positive, False Negatives, and True Negative based on tool results and ground truth, as shown in Table IV. It reports five standard metrics: Recall, Specificity, Precision, Accuracy, and F1 score to evaluate the quality of tools. We also report tool support rate (TSR), which is the ratio of how many test files are supported by a tool. The F1 score is a measure combining both precision and recall It provides a single metric that weights precision and recall in a balanced way, requiring both to have higher values for the F1-score value to rise. The reported adjusted F1 score, F1 score multiplied by the TSR, can show the true ability of a tool.

Table V shows the baseline results using the original coarse-grain evaluation strategy. For C/C++, Coderrect generated the best adjusted F1 score of 0.911. ThreadSanitizer performed the best for Fortran, with its adjusted F1 score of 0.889.

The high-precision evaluation method introduced in this paper considers more detailed information that is available in the data race tool reports. As shown in Table I, all selected tools can provide source line information in their data race reports. However, we observed that most of the selected tools are not ready to provide precise column information. Therefore, the first level of evaluation includes the source line information that is commonly available from all the tool reports. For the second level of evaluation, we additionally consider the

Tool Result	Ground Truth		Recall	Specificity	Precision	Accuracy	F1 Score
	True	False					
True	TP	FP	TP / (TP + FN)	TN / (TN + FP)	TP / (TP + FP)	(TP+TN) / (TP + FP + TN+ FN)	2 * (P * R) / (P + R)
False	FN	TN					

TABLE IV: Definition of metrics (Recall, Specificity, Precision, Accuracy and F1 Score)
TP: true-positive; FP: false-positive; TN: true-negative; FN: false-negative; P: precision; R:recall

Tool	Languages	TP	FN	TN	FP	Recall	Specificity	Precision	Accuracy	TSR	Adjusted F1
Intel Inspector	C/C++	69	13	80	7	0.841	0.920	0.908	0.881	0.955	0.873
ROMP		62	16	56	13	0.795	0.812	0.827	0.803	0.831	0.810
ThreadSanitizer		69	17	87	3	0.802	0.967	0.958	0.886	0.994	0.873
Coderrect		72	12	87	2	0.857	0.978	0.973	0.919	0.977	**0.911**
LLOV		55	28	78	9	0.663	0.897	0.859	0.782	0.960	0.748
Intel Inspector	Fortran	65	16	66	10	0.802	0.868	0.867	0.834	0.946	0.833
ROMP		58	15	50	11	0.794	0.820	0.841	0.806	0.807	0.817
ThreadSanitizer		52	13	65	0	0.800	1.000	1.000	0.900	0.783	**0.889**
Coderrect		62	18	65	11	0.775	0.855	0.849	0.814	0.939	0.810
LLOV		40	39	65	11	0.506	0.855	0.784	0.677	0.945	0.615

TABLE V: Baseline results using the original file-level, coarse-grain evaluation method.

information about the variable access type. Results of level 1 and level 2 evaluations are in Table VI and VII respectively.

It is notable that several key quality metrics (including Precision, Accuracy, and F1) reported in the Level 1 evaluation are significantly lower than the corresponding values reported in the previous coarse-granularity evaluation. The best adjusted F1 score for C/C++ now is Thread Sanitizer's 0.752 compared to the corresponding baseline score of 0.911. The reason is that more false positives and false negatives are reported. For example, 25 false positives are reported for Intel Inspector compared to the original 7. For Fortran, the best adjusted F1 score is Intel Inspector's 0.575 compared to 0.889. The Level 2 evaluation further requires the tools to report more precise information for the reported races about variable access types. A tool needs to pass even more information checking to generate final metrics. Even lower metric results are observed in the Level 2 evaluation compared to the results in the Level 1 evaluation. For example, the best adjusted F1 score for C/C++ is reduced to 0.715 while the score for Fortran is reduced to 0.497.

By exposing more false positives and false negatives, the high-precision evaluation process is able to identify more issues of a tool. For example, DRB016 has two expected races as shown in Listing 9. Intel Inspector detects the two expected races in RDB016 correctly. But it also reports an additional write/write race with both variable accesses at line 73. The previous file-level coarse-grain evaluation overlooks this issue and counts the entire report as a true positive. Using the new high-precision evaluation method, this additional write/write race is correctly counted as false-positive by both the Level 1 and Level 2 evaluation.

Listing 9: C: DRB016-outputdep-orig-yes.c

```
/*
Data race pairs: we allow two pairs to preserve the
    original code pattern.
 1. x@73:12:R vs. x@74:5:W
 2. x@74:5:W vs. x@74:5:W
*/

#include <stdio.h>
int a[100];

int main()
{
  int len=100;
  int i,x=10;

#pragma omp parallel for
  for (i=0;i<len;i++)
  {
    a[i] = x;
    x=i;
  }
  printf("x=%d",x);
  return 0;
}
```

C. Similarity analysis experiments

Among the microbenchmarks in DRB, 653 code regions are extracted by the OpenMP Extractor. These code regions include loop bodies and basic blocks that have OpenMP directives specified in the source codes. A total of 426,409 (653×653) cosine distance values can be generated but we only consider distances reported by distinct pairs of code regions and eliminate the distances from self-to-self comparisons. A total of 213,531 cosine values, $((1 + 652) \times 652)/2$, are reported in Table VIII. Two threshold values, 0.87 and 0.5 representing cosine values for 30° and 60°, are picked to categorize the levels of similarity.

There are 14 pairs of identical code regions reported. These identical pairs come from the following groups: First, many of them are from microbenchmarks with fixed size input data and variable size input data, for example DRB001-antidep1-orig-yes.c and DRB002-antidep1-var-yes.c. Second, some statements in the code region are highly identical but with minor differences in the array subscript. For example, DRB001-antidep1-orig-yes.c and DRB029-truedep1-orig-yes.c are considered identical because of the statements

Tool	Languages	TP	FN	TN	FP	Recall	Specificity	Precision	Accuracy	TSR	Adjusted F1
Intel Inspector	C/C++	76	18	149	25	0.809	0.856	0.752	0.840	0.955	0.744
ROMP		59	35	138	40	0.628	0.775	0.596	0.682	0.831	0.508
ThreadSanitizer		64	30	163	11	0.681	0.937	0.853	0.811	0.994	**0.752**
Coderrect		62	32	160	14	0.660	0.920	0.816	0.787	0.977	0.712
LLOV		33	61	143	46	0.351	0.757	0.418	0.540	0.960	0.367
Intel Inspector	Fortran	62	30	173	50	0.674	0.776	0.554	0.746	0.946	**0.575**
ROMP		48	44	172	56	0.522	0.754	0.462	0.659	0.807	0.395
ThreadSanitizer		45	47	193	29	0.489	0.869	0.608	0.719	0.783	0.424
Coderrect		44	48	194	28	0.478	0.874	0.611	0.717	0.939	0.504
LLOV		26	66	197	25	0.283	0.887	0.510	0.637	0.945	0.344

TABLE VI: Level 1: evaluation with source line information included

Tool	Languages	TP	FN	TN	FP	Recall	Specificity	Precision	Accuracy	TSR	Adjusted F1
Intel Inspector	C/C++	83	20	147	58	0.806	0.717	0.589	0.747	0.955	0.649
ThreadSanitizer		64	39	190	11	0.621	0.945	0.853	0.786	0.994	**0.715**
LLOV		33	65	170	46	0.337	0.787	0.418	0.558	0.960	0.358
Intel Inspector	Fortran	63	35	217	79	0.643	0.733	0.444	0.711	0.946	**0.497**
ThreadSanitizer		38	60	151	200	0.388	0.430	0.160	0.399	0.783	0.177
LLOV		26	71	265	25	0.268	0.914	0.510	0.686	0.945	0.332

TABLE VII: Level 2: evaluation using variable access types and source line information

`a[i]=a[i+1]+1;` and `a[i+1]=a[i]+1;` in the source codes respectively. Third, 006-indirectaccess2-orig-yes.c, 007-indirectaccess3-orig-yes.c and DRB008-indirectaccess4-orig-yes.c are identical by design to detect data races when different numbers of threads are used in the testing. The only differences are their different input data element values. Fourth, code regions have the same statements but in different order, for example DRB020-privatemissing-var-yes.c and DRB035-truedepscalar-orig-yes.c. And last, code regions that have only a function call statement can be treated identical because the OpenMP Extractor does not inspect the function definition that is called from the OpenMP code region.

The previous study reported about 45% of the pairs are considered identical, 32% are moderately identical and 23% are distinct. With additional static information included into the feature vectors, the evaluation reports about 54% of the pairs are considered highly identical, 44% are moderately identical and only 2.9% are distinct. Manual inspections were performed to investigate the differences between pairs in a highly identical group and pairs in a moderately identical group. Those identified as moderately identical are likely to have one code region coming from microbenchmarks in PolyBench. These code regions in PolyBench are generated by the polyhedral transformations and have more complex loop statements. A code region with loop statement and another code region without loop statement in it are categorized as moderately identical. Those pairs identified as distinct have more differences reflecting the fields in the feature vector. They tend to have different directives and clauses. In addition, they have more obvious differences in the source codes. For example, one has a loop statement but the other one does not.

In summary, the enhanced similarity analysis is helpful to guide detailed investigation of potential redundant code regions in DRB. However, including even more source code statistical information into feature vectors does not necessarily improve the distance-based similarity analysis to distinguish the differences among code regions. The cosine similarity is simply the cosine of the angle between two vectors. This feature poses the main drawback of the cosine distance that the magnitude of vectors is not taken into account. For the feature vector used in this work, there will be a higher difference reflected by cosine distance when there are more zero v.s. non-zero differences. Difference between two non-zero values does not seem to cause high impact in the similarity analysis. This can be seen from one example found in the result that a pair is more likely to be different when one block has a loop statement and the other one has not. Their feature vectors will have more zero v.s. non-zero differences: differences in directive, clause, having comparison operator, and having $++$ operator for index increment. A future study is required to discover a better similarity analysis approach. The directions include finding a more suitable distance measure that can handle high-dimensional vectors and reflect the value differences represented among vectors, or adding coefficient values into the feature vector elements for the distance measurement.

Cosine Distance	Degree	Similarity Index	Pair Number
[1.0]	0°	Identical	14
[0.87-1.0]	0°-30°	Highly Identical	114,086
[0.5-0.87)	30°-60°	Moderately Identical	92,646
[-1-0.5)	60°-180°	Distinct	6,132
Total			212,878

TABLE VIII: Results of the enhanced similarity analysis

VI. Related Work

Most OpenMP benchmarks focus on performance evaluation benchmarks. Popular examples include NPB [13], SPEComp [14], and SPEC ACCEL OpenMP version [15]. At the same time, several benchmarks for data race detection exist for the Java language. For example, JBench [6] is one of the popular data race benchmarks with 48 JAVA test cases and three data race tools support. A run-time error

detection (RTED) testing suite collects over ten-thousand runtime error tests including OpenMP test cases. Different from , RTED supports evaluation for system software's capability in runtime-error detection [16]. DataRaceBench [17] is the first specially designed benchmark suite for extensive OpenMP programs' data race detection. Its previous version, DataRaceBench v1.3.0 [8], incorporates Fortran support and more code patterns in general. In this paper, we enhance DRB further to have more accurate correctness checking, static tool support, as well as an improved similarity analysis.

Code similarity analysis is widely used in many different domains. For example, a recent paper [18] gives a good survey about binary code clone detection techniques, including those based on metrics, text, tokens, trees, and graphs. Deep learning is also used to measure code similarity. Deepsim [19] encodes code control flow and data flow into a semantic matrix to develop a model that measures code functional similarity for Java programs. The work of Statistical Similarity of Binaries [20] divides code into smaller fragments to compare the resemblance of procedures in stripped binaries. The Tracelet-Based Code Search in Executables [21] introduced a new static method to find similar functions in the code-base. The authors divided the function into tracelets, which are short, continuous, partial traces of execution. After that, they calculated the tracelet match score by aligning the tracelet with LCS variation (edit distance). These two methods analyze the binary code similarity by a similar approach. Our similarity analysis encodes both static and dynamic information of OpenMP code regions. A compiler is used to extract static code features. The dynamic information is derived from the results of state-of-the-art dynamic analysis tools.

VII. Conclusion

In this paper, we present the work to improve DataRaceBench to have a high-precision evaluation method to check the correctness of reports generated by different data race detection tools. This method includes a novel format to encode ground truth of data races in source code and several steps to postprocess the generated tool reports in order to enable different levels of correctness comparison. The test harness of DataRaceBench has also been improved to support static tools. Finally, we developed an improved similarity analysis. The experiment results show that the high-precision evaluation method is effective to better identify limitations of selected data race detection tools. The enhanced similarity analysis also is able to guide us to investigate similar code regions in DataRaceBench in more detail.

Future work will include collaboration with tool developers to file bug reports based on the new false positives and false negatives exposed by DataRaceBench. We will also explore more advanced similarity analysis algorithms and use them to guide the addition of new code patterns into DataRaceBench.

References

[1] S. Thayer, G. L. Gopalakrishnan, I. Briggs, M. Bentley, D. H. Ahn, I. Laguna, and G. L. Lee, "Archergear: data race equivalencing for expeditious hpc debugging," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020, pp. 425–426.

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

[3] A. Schmitz, J. Protze, L. Yu, S. Schwitanski, and M. S. Müller, "Dataraceonaccelerator–a micro-benchmark suite for evaluating correctness tools targeting accelerators," in *European Conference on Parallel Processing*. Springer, 2019, pp. 245–257.

[4] U. Bora, S. Das, P. Kureja, S. Joshi, R. Upadrasta, and S. Rajopadhye, "Llov: A fast static data-race checker for openmp programs," *arXiv preprint arXiv:1912.12189*, 2019.

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

[6] J. Gao, X. Yang, Y. Jiang, H. Liu, W. Ying, and X. Zhang, "Jbench: a dataset of data races for concurrency testing," in *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018, pp. 6–9.

[7] M. Jasper, M. Mues, M. Schlüter, B. Steffen, and F. Howar, "Rers 2018: Ctl, ltl, and reachability," in *International Symposium on Leveraging Applications of Formal Methods*. Springer, 2018, pp. 433–447.

[8] G. Verma, Y. Shi, C. Liao, B. Chapman, and Y. Yan, "Enhancing dataracebench for evaluating data race detection tools," in *2020 IEEE/ACM 4th International Workshop on Software Correctness for HPC Applications (Correctness)*. IEEE, 2020, pp. 20–30.

[9] F. Ye, M. Schordan, C. Liao, P.-H. 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)*. IEEE, 2018, pp. 42–50.

[10] B. Swain, Y. Li, P. Liu, I. Laguna, G. Georgakoudis, and J. Huang, "Ompracer: A scalable and precise static race detector for openmp programs," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–14.

[11] U. Bora, S. Das, P. Kukreja, S. Joshi, R. Upadrasta, and S. Rajopadhye, "Llov: A fast static data-race checker for openmp programs," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 4, pp. 1–26, 2020.

[12] "Coderrect," https://coderrect.com/.

[13] "NAS Parallel Benchmarks 3.0," https://github.com/benchmark-subsetting/NPB3.0-omp-C.

[14] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady, "Specomp: A new benchmark suite for measuring parallel computer performance," in *International Workshop on OpenMP Applications and Tools*. Springer, 2001, pp. 1–10.

[15] G. Juckeland, O. Hernandez, A. C. Jacob, D. Neilson, V. G. V. Larrea, S. Wienke, A. Bobyr, W. C. Brantley, S. Chandrasekaran, M. Colgrove *et al.*, "From describing to prescribing parallelism: Translating the spec accel openacc suite to openmp target directives," in *International Conference on High Performance Computing*. Springer, 2016, pp. 470–488.

[16] G. R. Luecke, J. Coyle, J. Hoekstra, M. Kraeva, Y. Xu, M.-Y. Park, E. Kleiman, O. Weiss, A. Wehe, and M. Yahya, "The importance of run-time error detection," in *Tools for High Performance Computing 2009*. Springer, 2010, pp. 145–155.

[17] C. Liao, P.-H. Lin, J. Asplund, M. Schordan, and I. Karlin, "Dataracebench: a benchmark suite for systematic evaluation of data race detection tools," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–14.

[18] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, and B. Maqbool, "A systematic review on code clone detection," *IEEE access*, vol. 7, pp. 86 121–86 144, 2019.

[19] G. Zhao and J. Huang, "Deepsim: deep learning code functional similarity," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 141–151.

[20] Y. David, N. Partush, and E. Yahav, "Statistical similarity of binaries," *ACM SIGPLAN Notices*, vol. 51, no. 6, pp. 266–280, 2016.

[21] Y. David and E. Yahav, "Tracelet-based code search in executables," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 349–360, 2014.