A Semantics-Driven Approach to Improving DataRaceBench#s OpenMP Standard Coverage

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

May 3, 2018

The 14th International Workshop on OpenMP 2018
Barcelona, Spain
September 27, 2018 through September 28, 2018

A Semantics-Driven Approach to Improving DataRaceBench's OpenMP Standard Coverage

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

Center for Applied Scientific Computing, Lawrence Livermore National Laboratory
Livermore, CA 94550, USA
{liao6,lin32,schordan1,karlin1}@llnl.gov

Abstract. DataRaceBench is a benchmark suite designed to systematically and quantitatively evaluate the effectiveness of data race detection tools. Its initial release in 2017 contained 72 C99 microbenchmarks with and without data races and was successfully used to evaluate several popular data race detection tools.

In this paper, we describe a novel semantics-driven approach to improving DataRaceBench's OpenMP standard coverage. Based on a traditional definition of data races, we define several semantic categories for parallelism, data-sharing attributes, and synchronization. This allows us to assign semantic labels to constructs, clauses and data-sharing rules in the OpenMP 4.5 specification. Based on these labels we then analyze the coverage of the initial release of DataRaceBench and add 44 new C and C++ microbenchmarks to improve the OpenMP standard coverage. Finally, we re-evaluate two popular data race detection tools with the new microbenchmarks, and show that the new version of DataRaceBench gives new insights about the selected tools.

1 Introduction

Benchmarks are widely used in many research communities to measure and assess research and development results 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. For example, the SPEC (Standard Performance Evaluation Corporation) [1] and LINPACK [8] play important roles in the high performance computing (HPC) community for performance improvements.

In the HPC community, data race bugs are notoriously damaging while extremely difficult to detect. We have developed a dedicated OpenMP benchmark suite, DataRaceBench [9], to help systematically and quantitatively evaluate data race detection tools for their strengths and limitations. The initial release in 2017, version 1.0.1 of DataRaceBench, included a set of OpenMP microbenchmarks with and without data races. It contained 72 C99 microbenchmarks and was used to generate detailed accuracy reports for four popular data race detection tools[3,5,6,2].

In this paper, we present a novel semantics-driven approach to analyzing and improving the OpenMP standard coverage of DataRaceBench by examining semantics of a data race. This process involves categorizing semantic categories related to data races, identifying and labeling OpenMP constructs, clauses and data-sharing attribute rules related to these semantic categories, analyzing coverage of existing microbenchmarks with respect to the semantic labels, and finally adding new microbenchmarks to improve coverage. Using this approach, we have added 44 new C and C++ microbenchmarks to DataRaceBench v1.2.0. We used the new version of DataRaceBench to re-evaluate two popular data race detection tools and discovered new insights.

The remainder of this paper is organized as follows. Section 2 gives an overview of the original DataRaceBench. Section 3 describes semantic analysis of data races and how we generate semantic labels for the OpenMP 4.5 specification. Coverage analysis and improvements are described in Section 4. Section 5 shows evaluation results. Section 6 presents the conclusion and future work.

2 Original DataRaceBench

DataRaceBench is a dedicated OpenMP benchmark suite to evaluate data race detection tools. The goal of this 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.

As shown in Figure 1, the initial release (v.1.0.1) of DataRaceBench contains 72 microbenchmarks written in C99. There are 40 microbenchmarks with known data races. They are called race-yes programs. The other 32 microbenchmarks are called race-no programs which are data race free. To enable scalable experiments, some race-yes programs use C99 variable-length arrays to allow user-specified input sizes as command line options.

Two scripts are also provided to run the benchmark suite and generate reports.

Several design guidelines are followed when creating microbenchmarks for DataRaceBench. The guidelines include:

- Each microbenchmark should be as small as possible to represent a pattern with and without data race. For example, there are programs demonstrating the use of one or more OpenMP constructs or a common parallel computing pattern (for example, reduction, stencil, indirect array accesses, etc.).
- Each microbenchmark program has a main function to support dynamic data race detection.
- We pair up race-yes programs with race-no programs, when necessary.
- If possible, a race-yes program 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 consolidates multiple runtime data races caused by the same pair of source code locations, into one data race.

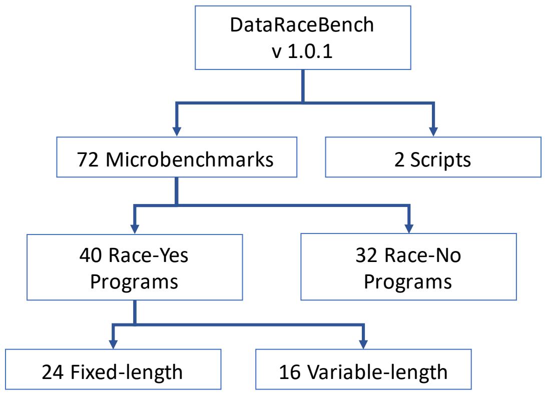

Fig. 1: Overview of initially released DataRaceBench Version 1.0.1

Figure 2 and Figure 3 show a pair of race-yes and race-no programs included in DataRaceBench. The first program has a pair of source code locations (two references to variable x at line 5) which will trigger data races. The reason is that there is loop-carried output dependence caused by the writes to the shared variable x within a parallel region. The second program fixes the data race bug by introducing a data-sharing clause, **lastprivate**, to make the accesses to x private within the region and copy its local value within the last iteration to its corresponding original variable after the end of the region.

```
//...
int i,x;
#pragma omp parallel for
for (i=0;i<100;i++)
{    x=i;  }
printf("x=%d",x);
```

Fig. 2: Race-yes example

```
//...
int i,x;
#pragma omp parallel for lastprivate(x)
for (i=0;i<100;i++)
{    x=i;  }
printf("x=%d",x);
```

Fig. 3: Race-no example

Using a data race detection tool to analyze a microbenchmark of DataRaceBench will generate several possible results. If the analysis tool detects an existing data race it is called a *true-positive (TP)*. If the tool reports a data race for a given program, but de-facto the data race does not exist, it is called a *false-positive (FP)* analysis result. Similarly, we can have *true-negative (TN)* and *false-negative(FN)* results. With the numbers of positives and negatives reported by the tool, several standard metrics, including *precision (P)*, *recall (R)* and accuracy (A), can be calculated. They are defined as follows: $P = TP/(TP+FP)$, $R = TP/(TP+FN)$, and $A = (TP+TN)/(TP+FP+TN+FN)$. More details of DataRaceBench can be found in a previous paper [9].

3 Semantic Analysis of Data Races

In order to discover what should be included in DataRaceBench, we study the semantics of a traditional definition [13] of data races, i.e., "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." Based on this definition, the occurrence of a data race depends on satisfying conditions related to at least five kinds of semantics: parallel (or concurrent), shared, variable, read/write access, and synchronization. As a preliminary study, we only focus on parallel, shared and synchronization semantics and examine the relevant C/C++ OpenMP constructs, clauses, and data-sharing attribute rules defined in the latest OpenMP 4.5 specification.

3.1 Parallel Semantics

We define the parallel semantics as the information indicating if a code region will be executed concurrently or not. Based on this definition, we categorize 26 directives (including their combined variants) specified in OpenMP 4.5 into this semantic category. They include **parallel**, **for**, **sections**, **single**, **master**, **simd**, **for simd**, **task**, **taskloop**, **taskloop simd**, **parallel for**, **parallel sections**, **target parallel**, **target teams** and so on. For example, the **sections** construct contains a set of structured blocks that are to be distributed among and executed by the threads in a team. It implies concurrent execution. Similarly, the **taskloop** construct specifies loop iterations will be executed in parallel using OpenMP tasks. Its semantics literally has the word of parallel. Yet another example is the **master** construct, which specifies a structured block that is executed by the master thread of the team. It indicates the region will not be executed concurrently, but by a single thread. Some clauses are also related to parallel semantics. They include **if**, **num_threads**, **collapse** and **num_teams**.

To facilitate coverage analysis, we assign a semantic label (SID) for each relevant directive or clause. The directives related to parallel semantics are labeled as PD01 through PD26. The clauses are labeled as PC01 through PC04.

3.2 Shared Semantics

We define shared semantics as any information describing if a variable is visible and accessible by multiple threads or not. OpenMP 4.5 uses an entire subsection (Sec. 2.15) to describe its data environment, including data-sharing attribute rules and clauses (Sec. 2.15.1). The high-level logic flow of the subsection is shown in Figure 4. The decision about a variable's data-sharing attribute starts with a question (D1) about if a variable is referenced in some eligible OpenMP regions (dynamic instances of OpenMP code blocks) including **target**, **teams**,**parallel**, **simd**, task generating (**task**, **taskloop**) and worksharing (**for**, **sections**, **single**, and **workshare**). Only a variable referenced in some regions is interesting and checked against the second question (D2): Is the variable referenced in a construct (the lexical extent of an executable directive[1])? If the answer is no, a set of not-in-construct rules apply (defined in Sec. 2.15.1.2 in OpenMP 4.5). If yes, three types of rules apply (defined in Sec. 2.15.1.1 in OpenMP 4.5): predetermined, implicitly determined, or explicitly determined.

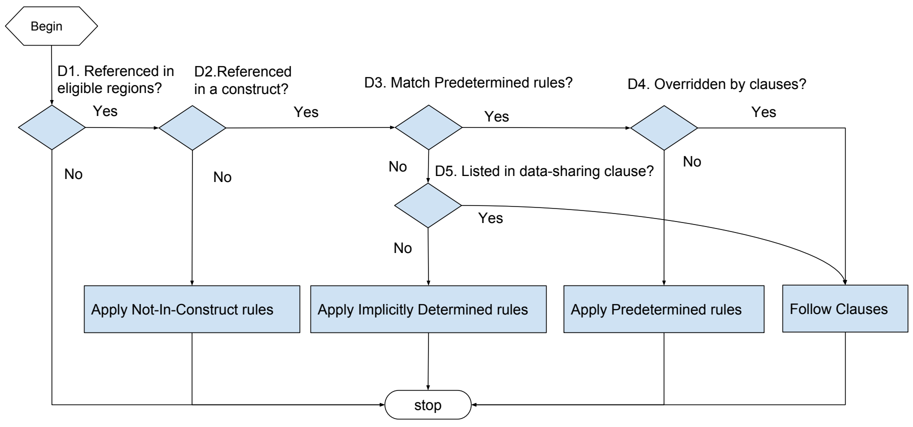
Begin
D1. Referenced in eligible regions?
D2.Referenced in a construct?
D3. Match Predetermined rules?
D4. Overridden by clauses?
D5. Listed in data-sharing clause?
Yes
No
Apply Not-In-Construct rules
Apply Implicitly Determined rules
Apply Predetermined rules
Follow Clauses
stop

Fig. 4: Flowchart of the data-sharing attribute rules

Rules for Not Referenced in a Construct. OpenMP 4.5 uses six sentences to describe when a variable is not referenced in a construct. We label them as NIC1 through NIC6 based on the order the sentences appear in the specification. Since the order of the rules in the specification is rather ad-hoc, we reorganize them as follows:

- Declared inside the called routine
  - NIC1: if the variable uses static storage, it is shared
  - NIC6: otherwise, it is private
- File-scope or namespace-scope variable
  - NIC2.1: threadprivate if the variable is in a threadprivate directive
  - NIC2.2: shared otherwise
- Function arguments in C++
  - NIC5.1: same as actual arguments if passed by reference
  - NIC5.2: private if passed by values (not explicitly listed in OpenMP)
- Dynamic storage:
  - NIC3 - objects with dynamic storage duration are shared.
- Static data members
  - NIC4.1: threadprivate if within a threadprivate directive

[1]In OpenMP, an executable directive is a directive that is not declarative. It may be placed in an executable context.

  • NIC4.2: shared otherwise

We split NIC2 into two sub rules (NIC2.1 and NIC2.2) since the original sentence checks a condition and leads to two different data-sharing attributes. For coverage analysis, it is better to have separated rules for different data-sharing attributes. Similarly, NIC4 is split into NIC4.1 and NIC4.2. NIC5.1 only states what happens when function arguments use pass-by-reference. We think NIC6 does not really cover function arguments passed by values, since a function argument is different from a variable declared inside a function body. We added NIC5.2 to indicate a function argument passed by value should be private to be consistent with other rules.

Rules for Predetermined Attributes. The rules for predetermined attributes (prefix PDT) are summarized below. As with the NIC rules we perform similar rule re-organization and splitting. For example, the original PDT5 rule is related to a loop iteration variable associated with for-loops of four types of constructs. We split it into four rules: one for each construct.

- Declared in a scope inside the construct
  • PDT2: private if the variable has an automatic storage duration
  • PDT8: shared if the variable has an static storage duration
- Declared in a scope outside of the construct
  • PDT1: threadprivate if within a threadprivate directive
- Dynamic storage: PDT3 - shared if the variable has a dynamic storage duration
- Static data member: PDT4 - shared if the variable is a static data member
- If loop iteration variables are in question:
  • PDT5.1: private if in the associated for-loops of a for construct
  • PDT5.2: private if in the associated for-loops of a parallel for construct
  • PDT5.3: private if in the associated for-loops of a taskloop construct
  • PDT5.4: private if in the associated for-loops of a distribute construct
  • PDT6: linear if the loop is the only loop associated with the SIMD construct
  • PDT7: lastprivate if there are multiple loops associated with the SIMD construct
- Array section: PDT9 - firstprivate if the variable is an array section mapped within a target construct, and derived from a variable of a pointer type.

Note that unlike many NIC rules stating two choices for a condition (e.g. NIC1 and NIC6, NIC2.1 and NIC2.2), most PDT rules (e.g. PDT1, PDT5.1 through 5.4, PDT6, etc.) only state what will happen when certain conditions are met. When these conditions are not met, the decision will be deferred to a later stage using either implicitly determined rules or explicit data-sharing clauses.

Implicitly Determined Rules. We label the seven sentences for implicitly determined rules with IDs and re-organized them as follows:

- Default clause: IDT1 - for variables in a parallel, teams, task generating constructs, follow the default clause if it is present
- In a Parallel construct: IDT2 - the variables are shared if no default clause is present.
- In a Target construct:
  - IDT4.1: variables that are not mapped are firstprivate.
  - IDT4.2: variables that are mapped, follow data-mapping attribute rules and clauses.
- Task generating construct:
  - IDT5: In an orphaned task generating construct, formal arguments passed by reference are firstprivate
  - IDT6: A variable is shared when it is in a task generating construct without a default clause, its data sharing attribute is not determined by the above rules, and the same variable in the enclosing context is determined to be shared by all implicit tasks bound to the current team.
  - IDT7: In a task generating construct, a variable without applicable rules above is firstprivate.
- Others: IDT3 - In constructs other than task generating or target constructs (e.g. teams, simd and worksharing), these variables reference the variables with the same names that exist in the enclosing context, if no default clause is present.

Explicit Data-Sharing Clauses. Finally, there are seven clauses indicating data-sharing attributes, including **default**, **shared**, **private**, **firstprivate**, **lastprivate**, **reduction** and **linear**. We categorize them into a DSC (data-sharing clause) set (DSC01 through DSC07).

3.3 Synchronization Semantics

We define synchronization semantics as any information deciding if there is any synchronization mechanism to prevent the shared accesses to a variable from being simultaneous or not. We categorize the following OpenMP directives and clauses as relevant to synchronization, including **nowait**, **critical**, **barrier**, **taskwait**, **taskgroup**, **atomic**, **flush**, **ordered** (both clause and directive) and **depend**. They are labeled as N01 through N10. N00 is reserved to indicate that no explicit synchronization is specified.

4 Coverage Analysis and Improvements

For each semantic label, if there is a microbenchmark using the corresponding construct, clause or rule, we claim that the label is covered in our coverage analysis. For example, a microbenchmark shown in Figure 2 covers PD12 (**parallel for**), PDT5.1 (predetermined to be private for an associated loop iteration variable) and N00 (no explicit synchronization is specified).

4.1 Analysis Methods

Some coverage information can be obtained by checking if some OpenMP keywords (such as **collapse**, **depend**, and **taskgroup**) are used in our benchmark suite. This gives us an overview of which semantic labels are covered and which are missing.

To recognize complex code patterns beyond keywords, we built a simple source analysis tool, namely CoverageAnalyzer, using the ROSE source-to-source compiler framework [7,10]. CoverageAnalyzer parses source files into Abstract Syntax Trees and finds code patterns satisfying conditions defined in data-sharing attribute rules. For example, to check if PDT8 is covered, Coverage-Analyzer tries to find all OpenMP regions first, then searches each region for locally declared variables. If the variable is not declared static, we find a match to the conditions corresponding to PDT8 and conclude that PDT8 is covered.

Sometimes we got lucky and did not have to implement condition search for all rules in CoverageAnalyzer. For example, all NIC rules require a code pattern in which a variable is referenced within an OpenMP region, but not within an OpenMP construct. This can only happen through a function call. CoverageAnalyzer finds that none of the existing programs in v1.0.1 has an OpenMP region in which a function call to user-defined functions is made. So we can safely conclude that none of NIC rules are covered.

4.2 Analysis Results

The coverage of semantic labels in each semantic category is summarized in Table 1. In the parallel category, missed constructs include **master**, **taskloop**, **teams** and their applicable combined directives. Within the shared semantic category, NIC rules have zero coverage while two data-sharing clauses (**default** and **linear**) in DSC are not covered. For PDT and IDT, uncovered rules include those involving static variables, **threadprivate**, **collapse**, **taskloop**, **distribute**, multiple loops associated with SIMD, orphaned task constructs using formal arguments passed by reference and so on. For synchronization semantics, only two out of ten relevant clauses are covered (**nowait** and **depend**).

	Parallel	**Shared**				**Sync.**
		NIC	**PDT**	**IDT**	**DSC**	
Semantic Label Count	30	9	12	8	7	10
Covered Labels	PD1-4,6,8,11 12,14,15,PC02		2,3,5.1 5.2,6	2,3,4.1,6	2-6	1,10
Covered Label Count	11	0	5	4	5	2
Coverage Ratio	36.67%	0.0%	41.67%	50.0%	71.43%	20.0%

Table 1: Coverage Analysis Result for v1.0.1 of DataRaceBench

4.3 Improving Coverage

Based on the coverage analysis results, we added 44 new microbenchmarks into DataRaceBench Version 1.2.0 [2] to cover the missed semantic labels. For simplicity, we treat some combined constructs (e.g. **target simd**) as covered if their individual constructs are covered by existing microbenchmarks. For example, Figure 5 shows a new microbenchmark program to cover NIC4.1 and NIC4.2. In the case of not referenced within a construct, a static data member should be shared, unless it is within a **threadprivate** directive. Figure 6 covers both **ordered** clause and directive. **ordered**(2), an OpenMP 4.5 addition, also associates two loops and make their loop iteration variables private. **target teams** and **taskgroup** are covered in Figure 7 and Figure 8 respectively.

```
class A {
public:
  static int ctr;
  static int pctr;
#pragma omp threadprivate(pctr)
};
int A::ctr=0;
int A::pctr=0;
A a;
void foo()
{
   a.ctr++;
   a.pctr++;
}
int main()
{
#pragma omp parallel
      foo();
//...
}
```

Fig. 5: race-yes using static data members

```
#include <stdio.h>
int a[100][100];
int main()
{
  int i, j;
#pragma omp parallel for ordered(2)
  for (i = 0; i < 100; i++)
    for (j = 0; j < 100; j++)
      {
        a[i][j] = a[i][j] + 1;
#pragma omp ordered depend(sink:i-1,j) \
              depend (sink:i,j-1)
        printf ("test␣i=%d␣j=%d\n",i,j);
#pragma omp ordered depend(source)
      }
  return 0;
}
```

Fig. 6: race-no using ordered(2)

As a result, DataRaceBench Version 1.2.0 covers all semantic labels from each semantic group. This means that the new coverage ratios are all equal to 100%, as shown in Table 2.

	Parallel	**Shared**				**Sync.**
		NIC	**PDT**	**IDT**	**DSC**	
Semantic Label Count	30	9	12	8	7	10
Covered Labels	all	all	all	all	all	all
Covered Label Count	30	9	12	8	7	10
Coverage Ratio	100%	100%	100%	100%	100%	100%

Table 2: Coverage Analysis Result for v1.2.0 of DataRaceBench

[2] Available at `https://github.com/LLNL/dataracebench/releases`

```
// ...
  double a[len];

  /*Initialize with some values*/
  for (i=0; i<len; i++)
    a[i]= ((double)i)/2.0;

#pragma omp target map(tofrom: a[0:len])
#pragma omp teams num_teams(2)
  {
    a[50]*=2.0;
  }
```

Fig. 7: race-yes using target+teams

```
  int result = 0;
#pragma omp parallel
#pragma omp single
  {
#pragma omp taskgroup
#pragma omp task
      {
        sleep(3); result = 1;
      }
#pragma omp task
    result = 2;
  }
  assert (result==2);
```

Fig. 8: race-no using taskgroup

5 Evaluation

In order to assess if the new microbenchmarks in DataRaceBench v1.2.0 are beneficial, we use them to evaluate two popular data race detection tools, Archer and Intel Inspector. Archer [6] is an OpenMP data race detector that exploits ThreadSanitizer [5] to achieve scalable happens-before tracking. It uses static analysis to reduce false positives generated by the dynamic analysis performed by ThreadSanitizer. 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 [12,11], with added memory error checking. The versions of the selected tools used are listed in Table 3, with the compilers used with these tools (either to build the tools, compile the microbenchmarks, or both).

Tool	**Version**	**Compiler**
Archer	towards_tr4 branch	Clang/LLVM 4.0.1
Intel Inspector	2018 (build 522981)	Intel Compiler 18.0.1

Table 3: Data race detection tools: versions and compilers

Intel Inspector provides different levels of analysis with varying configurations. We configure the maximum level analysis in our evaluation using the command line: `inspxe-cl -collect ti3 -knob scope=extreme -knob stack-depth=16 -knob use-maximum-resources=true`.

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. We ran each tool 5 times for each microbenchmark using 72 threads. For each run, we use ten minutes as a timeout limit to terminate potential runtime hanging.

5.1 Experiment Results

Table 4 shows our experimental results. The first column lists the file names (each with a prefix such as DRB072 as a short ID) of all the newly added microbenchmark programs. The second column indicates if the program is known to contain a data race or not ('Y' or 'N'). During multiple runs for a given program, a tool may report different numbers of data races detected. So ranges of numbers (min race - max race) are given in Column 3 and 5 of the table. For example, an entry of 0-0 means that no data race was found in any run of the respective tool. An entry of 1-3 means in all five runs at least one data race was detected. If a range such as 0-4 is reported, this means a tool generated mixed results for a given program.

Column 4 and 6 (labeled as "type") give a verdict for a tool's result for a given program. Based on the range numbers, the result is given as true negative (TN), false positive (FP) or mixed TN and FP for a program without known data races. Similarly, a tool's result an be true positive (TP), false negative (FN) or mixed TP and FN for a race-yes program.

In some cases, a tool may fail due to errors during compilation or runtime steps. We mark the result as compile-time segmentation fault (CSF), unsupported feature by a compiler (CUN), runtime segmentation fault (RSF) or runtime timeout (RTO). If any error happens, we try to investigate log files to identify any valid true or false positives. Negative reports are ignored since a negative test report with errors is inconclusive. For example, a tool may trigger a runtime timeout and generate partial logs with identified data races, which should be counted. Table 5 summarizes the numbers of positive, negative and unknown (marked as not available or N/A) results based on the information in Table 4.

The results show that new benchmark programs generate new insights for the two tools. Archer did not report any false positives or false negatives in the experiments. However, 13 programs triggered the tool to have some compile-time or runtime errors. Five of these error happened because the version of Clang does not support the OpenMP 4.5 features used in DRB094, DRB095, DRB096, DRB100 and DRB112 (marked as CUN). Another five errors are compiler segmentation faults raised by a phase called InstrumentParallel, for DRB085, DRB086, DRB087, DRB091 and DRB102 (marked as CSF). Runtime segmentation faults happened for DRB097 and DRB116 (marked as RSF). A runtime timeout (RTO) happened with DRB106. The tool generated partial results with true positives for DRB106. We are actively working with the Archer developers to address these issues in their latest development branch.

In comparison, Intel Inspector reported mixed results (TN FP) for DRB096, a program using **taskloop** combined with **collapse**(2) to cover PDT 5.3. In only one out of the five runs, the tool reported a write-to-write race for loop iteration variables. The tool also generated two false positives (FP) for DRB105 and DRB107. DRB105 is a classic task implementation of Fibonacci number generation using **taskwait**. The tool reported a write-to-write data race for the line of i=fib(n−1); For DRB107 (shown in Figure 8 using **taskgroup**), the tool reported two tasks writing to result causing a data race. DRB094 (shown in Fig. 6) caused a runtime timeout error (hanging) for Intel Inspector. In this program, the 2nd loop is associated with **ordered**(2) so its loop interaction variable should be pri-

Microbenchmark Program	R	Data Race Detection Tools			
		Archer		Intel Inspector	
		min race - max race	type	min race - max race	type
DRB073-doall2-orig-yes.c	Y	84 - 92	TP	2 - 2	TP
DRB074-flush-orig-yes.c	Y	1 - 3	TP	1 - 1	TP
DRB075-getthreadnum-orig-yes.c	Y	71 - 71	TP	1 - 1	TP
DRB076-flush-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB077-single-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB078-taskdep2-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB079-taskdep3-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB080-func-arg-orig-yes.c	Y	71 - 71	TP	1 - 1	TP
DRB081-func-arg-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB082-declared-in-func-orig-yes.c	Y	71 - 71	TP	1 - 1	TP
DRB083-declared-in-func-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB084-threadprivatemissing-orig-yes.c	Y	71 - 71	TP	1 - 1	TP
DRB085-threadprivate-orig-no.c	N	-	CSF	0 - 0	TN
DRB086-static-data-member-orig-yes.cpp	Y	-	CSF	1 - 1	TP
DRB087-static-data-member2-orig-yes.cpp	Y	-	CSF	1 - 1	TP
DRB088-dynamic-storage-orig-yes.c	Y	71 - 71	TP	1 - 1	TP
DRB089-dynamic-storage2-orig-yes.c	Y	71 - 71	TP	1 - 1	TP
DRB090-static-local-orig-yes.c	Y	71 - 71	TP	1 - 1	TP
DRB091-threadprivate2-orig-no.c	N	-	CSF	0 - 0	TN
DRB092-threadprivatemissing2-orig-yes.c	Y	71 - 71	TP	1 - 1	TP
DRB093-doall2-collapse-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB094-doall2-ordered-orig-no.c	N	-	CUN	0 - 0	RTO
DRB095-doall2-taskloop-orig-yes.c	Y	-	CUN	2 - 2	TP
DRB096-doall2-taskloop-collapse-orig-no.c	N	-	CUN	0 - 4	FP TN
DRB097-target-teams-distribute-orig-no.c	N	0 - 0	RSF	0 - 0	TN
DRB098-simd2-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB099-targetparallelfor2-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB100-task-reference-orig-no.cpp	N	-	CUN	0 - 0	TN
DRB101-task-value-orig-no.cpp	N	0 - 0	TN	0 - 0	TN
DRB102-copyprivate-orig-no.c	N	-	CSF	0 - 0	TN
DRB103-master-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB104-nowait-barrier-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB105-taskwait-orig-no.c	N	0 - 0	TN	3 - 4	FP
DRB106-taskwaitmissing-orig-yes.c	Y	35 - 48	RTO TP	4 - 6	TP
DRB107-taskgroup-orig-no.c	N	0 - 0	TN	1 - 1	FP
DRB108-atomic-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB109-orderedmissing-orig-yes.c	Y	71 - 71	TP	1 - 1	TP
DRB110-ordered-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB111-linearmissing-orig-yes.c	Y	73 - 85	TP	1 - 2	TP
DRB112-linear-orig-no.c	N	-	CUN	0 - 0	TN
DRB113-default-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB114-if-orig-yes.c	Y	42 - 48	TP	1 - 1	TP
DRB115-forsimd-orig-yes.c	Y	44 - 47	TP	1 - 1	TP
DRB116-target-teams-orig-yes.c	Y	0 - 0	RSF	1 - 1	TP

Table 4: Evaluation report (column R: whether a program contains a data race)

Tool	Race:Yes				Race:No			
	TP	TP/FN	FN	N/A	TN	TN/FP	FP	N/A
Archer	15	0	0	4	17	0	0	8
Intel Inspector	19	0	0	0	21	1	2	1

Table 5: The numbers of positive, negative and unknown results of the tools

vate according to PDT5.1. Making j explicitly private will fix the hanging. We have reported these issues to Intel.

6 Conclusion

In this paper, we presented a semantics-driven approach to analyzing and improving DataRaceBench's coverage of the OpenMP standard. We focused on three semantic categories (parallel, shared and synchronization) and labeled a set of relevant OpenMP language constructs, clauses and rules for coverage analysis. The application of our approach resulted in adding 44 new microbenchmarks which significantly increased DataRaceBench's coverage. Finally, the new microbenchmarks were used to re-evaluate two data race detection tools: Intel Inspector and Archer. While these two tools performed almost equally well in our original evaluation [9], the new microbenchmarks reveal that Intel Inspector outperforms Archer in terms of supporting more microbenchmarks without any errors. However, there is still room for improvements for Intel Inspector when analyzing programs using **taskloop**, **taskwait** or **taskgroup**.

In addition, as an unexpected side effect of extracting semantics from the OpenMP 4.5 standard, we found a misuse of the term "construct". **declare simd** is called a construct while it is a non-executable declarative directive and an OpenMP construct must be an executable directive. We have reported this issue to the OpenMP language committee. Another discovery is that the data-sharing attribute rules in OpenMP are surprisingly difficult to understand. We had to reorganize these rules, split some of them, and made previously hidden rules explicit to extract semantic labels. We suggest to the OpenMP language committee to improve the clarity of the rules and define an official algorithm.

In the future, we plan to explore semantics related to variables and read-/write accesses. We also want to increase DataRaceBench's coverage of OpenMP runtime library routines and environment variables. In the domain of scientific computing, only a few computational patterns are covered in DataRaceBench, such as stencil and matrix multiplication. Adding more representative numerical computation patterns with and without data races may also be beneficial.

7 Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344, Lawrence Livermore National Security, LLC, via projects LDRD 17-ERD-023. IM release number LLNL-CONF-750770.

References

1. SPEC's Benchmarks (1995), `http://www.spec.org/benchmarks.html`
2. Helgrind (2000), `http://valgrind.org/docs/manual/hg-manual.html`
3. Intel inspector 2017 (2017), `https://software.intel.com/en-us/intel-inspector-xe`
4. Livermore computing quartz system (2017), `https://hpc.llnl.gov/hardware/platforms/Quartz`
5. Threadsanitizer (2017), `https://github.com/google/sanitizers`
6. Atzeni, S., Gopalakrishnan, G., Rakamaric, Z., Ahn, D.H., Laguna, I., Schulz, M., Lee, G.L., Protze, J., Müller, M.S.: Archer: effectively spotting data races in large openmp applications. In: Parallel and Distributed Processing Symposium, 2016 IEEE International. pp. 53–62. IEEE (2016)
7. Davis, K., Quinlan, D.: Rose: An optimizing preprocessor for the object-oriented overture framework. http:// www.c3.lanl.gov/ ROSE/
8. Dongarra, J.J., Luszczek, P., Petitet, A.: The linpack benchmark: past, present and future. Concurrency and Computation: practice and experience **15**(9), 803–820 (2003)
9. Liao, C., Lin, P.H., Asplund, J., Schordan, M., Karlin, I.: DataRaceBench: a benchmark suite for systematic evaluation of data race detection tools. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. p. 11. ACM (2017)
10. Liao, C., Quinlan, D.J., Panas, T., de Supinski, B.R.: A ROSE-Based OpenMP 3.0 Research Compiler Supporting Multiple Runtime Libraries. In: Sato, M., Hanawa, T., Müller, M.S., Chapman, B.M., de Supinski, B.R. (eds.) IWOMP. Lecture Notes in Computer Science, vol. 6132, pp. 15–28. Springer (2010)
11. Petersen, P., Shah, S.: Openmp support in the intel® thread checker. In: International Workshop on OpenMP Applications and Tools. pp. 1–12. Springer (2003)
12. Sack, P., Bliss, B.E., Ma, Z., Petersen, P., Torrellas, J.: Accurate and efficient filtering for the intel thread checker race detector. In: Proceedings of the 1st workshop on Architectural and system support for improving software dependability. pp. 34–41. ACM (2006)
13. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. **15**(4), 391–411 (Nov 1997). https://doi.org/10.1145/265924.265927, `http://doi.acm.org/10.1145/265924.265927`