LLNL-CONF-756830

Using Polyhedral Analysis to Verify OpenMP Applications are Data Race Free

F. Ye, M. Schordan, C. Liao, P. Lin, I. Karlin, V. Sarkar

August 21, 2018

Correctness 2018: Second International Workshop on Software Correctness for HPC Applications
Dallas, TX, United States
November 12, 2018 through November 12, 2018

Using Polyhedral Analysis to Verify OpenMP Applications are Data Race Free

Fangke Ye[1], Markus Schordan[2], Chunhua Liao[2], Pei-Hung Lin[2], Ian Karlin[3], Vivek Sarkar[1]
[1] School of Computer Science, Georgia Institute of Technology, Atlanta, USA
[2] Center of Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, USA
[3] Livermore Computing, Lawrence Livermore National Laboratory, Livermore, USA
Email: yefangke@gatech.edu, schordan1@llnl.gov, liao6@llnl.gov, lin32@llnl.gov, karlin1@llnl.gov, vsarkar@gatech.edu

***Abstract*—Among the most common and hardest to debug types of bugs in concurrent systems are data races. In this paper, we present an approach for verifying that an OpenMP program is data race free. We use polyhedral analysis to verify those parts of the program where we detect parallel affine loop nests. We show the applicability of polyhedral analysis with analysis-enabling program transformations for data race detection in HPC applications. We evaluate our approach with the dedicated data race benchmark suite DataRaceBench and the LLNL Proxy Application AMG2013 which consists of 75,000 LOC. Our evaluation shows that polyhedral analysis can classify 40% of the DataRaceBench 1.2.0 benchmarks as either data race free or having data races, and verify that 41 of the 114 (36%) loop nests of AMG2013 are data race free.**

***Index Terms*—OpenMP, Polyhedral Analysis, Data Race.**

I. Introduction

Data races are among the most common and hardest to debug types of bugs in concurrent systems. A data race occurs when two or more threads perform simultaneous conflicting data accesses to the same memory location without proper synchronization and at least one access is a write. Data race bugs may lead to unpredictable results of a parallel program even with the exact same input. Due to this behavior, the difficulties in detecting and the time required to fix data race bugs can greatly reduce productivity. Therefore, there is an increasing demand for data race detection tools and many industrial and research efforts provide various approaches to data race detection.

The DataRaceBench benchmark suite [15] was specifically designed for systematic evaluation of data race detection tools with a focus on the OpenMP parallel programming model. The development team of DataRaceBench presented initial evaluation results using quantitative metrics for four selected data race detection tools: Archer, ThreadSanitizer, Helgrind and Intel Inspector. The evaluation with DataRaceBench lead to the following discoveries:

- OpenMP awareness is a necessity for detecting data races in programs using the OpenMP programming model. Even if an OpenMP runtime library is implemented using Pthreads, a tool that only considers Pthread semantics will miss certain details of the OpenMP semantics that are checked at compile time, leading to false alarms (false positives) of the analysis.
- With all four of the evaluated dynamic data race detection tools, it is necessary to run the tool multiple times (with the exact same input) because some thread schedules can differ each time the program is executed, and some do result in a data race while others do not. Requiring multiple runs also increases the overall runtime of using dynamic analysis tools in practice.
- Dynamic testing tools can be sensitive to the number of threads used in testing. Some data races only occur when a certain number of threads are used. Testing code using a dynamic tool can require trying multiple thread counts to cover all possible data race bugs, further increasing the time needed to test a code.

Dynamic tools use testing to detect existing data races, but they cannot guarantee that a program is data race free. In contrast, static analysis can guarantee that a program is data race free whereas testing tools cannot. Our approach uses static analysis and the polyhedral model, a formalism developed for reasoning about program regions that use counted for-loops to manipulate dense arrays. Many applications of the polyhedral model have been found since its introduction [4]. Polyhedral analysis is being used also in optimizing compilers which have shown great promise in delivering high-performance by starting from a single input source and generating optimized code for a variety of target platforms (e.g. to map affine stencil computations to CPUs [5], [13], GPUs [10] and FPGAs [18]).

In this paper, we present a method for data race detection based on polyhedral analysis with analysis-enabling program transformations. We cannot verify all loops and OpenMP constructs with this single approach, but it allows us to carve out all those parts of a given program for which we can provide a conclusive result: that a given loop nest is data race free for an arbitrary iteration count of those verified loops. For the remaining loops and constructs in a program, other methods for data race detection can be used. Thus, the method should be used in combination with other verification techniques to verify the correctness of an entire program.

Our approach carves out parallel affine nested loops, which can be handled by a polyhedral analysis, and performs code transformations to enable polyhedral analysis of programs

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 and was supported by the LLNL-LDRD Program under Project No. 17-ERD-023. IM Release Nr. LLNL-CONF-756830.

which otherwise could not be handled. While the analysis can verify some of the loops in a program it cannot handle loops with indirect addressing or pointers. For program regions with for-loops and dense arrays, our polyhedral analysis shows high precision and no false positives are reported (= any data race that is reported does indeed exist). We also do not produce any false negatives which would indicate a bug in our implementation since it is designed to be sound. Programs that cannot be verified to be data race free, and for which we cannot detect a data race either, are reported as "unverified".

We perform an evaluation of our approach using DataRaceBench and demonstrate its applicability to real applications by applying it to the LLNL proxy application AMG2013[1]. We describe more details of the selected application in Sec. IV.

The contributions in this paper are:

- We present an approach for extending the applicability of polyhedral analysis with analysis-enabling program transformations. The presented transformations enable the application of polyhedral analysis when it could not be applied otherwise. This concerns the handling of private variables and out-of-bounds checks. The out-of-bounds check is necessary to also address cases in the C/C++ language where legal out-of-bounds accesses to elements inside a multi-dimensional array can cause data races.
- We determine whether a benchmark has a data race or does not for 40 % of the microbenchmarks in DataRaceBench [15] (current version 1.2.0 with 116 microbenchmarks). For those microbenchmarks without data races, we report how many microbenchmarks we can successfully verify to be data race free. For microbenchmarks that (intentionally) include data races, the polyhedral analysis reports conflicts and we report them as potential data races. If our pre-analysis determines that not all conditions for a correct polyhedral analysis hold for a given loop nest, we report it as unverified.
- We also apply our approach to the LLNL proxy application AMG2013. We can verify 41 of the 114 parallel loops to be data race free. This also involves a number of program transformation so that the polyhedral analysis can be applied. AMG2013 is an LLNL proxy application with 75,000 lines of code.

The rest of the paper is organized as follows. In Sec. II we give an overview of our approach and show some examples from the verified programs to illustrate some of the high-level aspects of our verification method. In Sec. III we describe our approach in detail. In Sec. IV we present an extensive evaluation of our approach, using DataRaceBench [15] and the LLNL proxy app AMG2013. In Sec. V we discuss related work before we conclude in Sec. VI.

II. Overview

The presented analysis technique is part of our work on the data race detection tool DRACO (Data Race Analysis and COrrectness) which is based on the ROSE compiler infrastructure[2]. One of the design goals of the tool is to analyze real applications without user interaction. The tool takes as input a C/C++ program and verifies that the program is data race free. In case we cannot verify the entire program, it reports the specific parts of the program that were verified. The granularity of reporting is functions and loops. Functions are reported by their respective names if the entire function can be verified. If an entire function cannot be verified then the OpenMP loop(s) (by their starting line number) that are verified within this respective function are reported. Parallel sections are reported data race free if the entire section is data race free. DRACO will be released as open source and distributed with ROSE when it is productized.

```
int main(int argc, char* argv[]) {
  int i;
  int a[2000];
  for (i=0; i<2000; i++)
    a[i]=i;
#pragma omp parallel for
  for (i=0;i<1000;i++)
    a[2*i+1]=a[i]+1;
  printf("a[1001]=%d\n", a[1001]);
  return 0;
}
```

Fig. 1: DataRaceBench benchmark DRB033 with a detected data race.

A. Examples

This section presents three examples that DRACO analyzed as either data race free or detected a data race in. Each example represents one of the three groups of programs that we analyze. Two examples are microbenchmarks from DataRaceBench, one with a known data race and one without a known data race. The third example is a code snippet of a verified loop from the LLNL proxy application AMG2013.

1) Data Race Bench - detected data race: In Fig. 1 we show the source code of the DataRaceBench microbenchmark DRB033. Variable `i` is implicitly private. This benchmark has a data race in the `omp parallel for` loop which is detected by our analysis and the line number of the data race is reported.

2) Data Race Bench DRB043 - verified to be data race free: The DataRaceBench microbenchmark DRB043[3] is a benchmark with no known data race. It is one of the benchmarks generated by PolyOpt[4]. In total it has 6 parallel loops. The iteration variable is implicitly private in each loop. Arrays A, B, and X are referred to by pointers passed as arguments to the function `kernel_adi`. Those array pointers refer to different arrays in the benchmark and no aliasing occurs. Aliasing also does not cause data races in other DataRaceBench benchmarks with affine loop nests.

3) AMG2013 - verified to be data race free: In Fig. 2 one of the many loops verified to be data race free in the proxy

[1]AMG2013: https://computation.llnl.gov/projects/co-design/amg2013

[2]ROSE: http://rosecompiler.org

[3]https://github.com/LLNL/dataracebench/blob/master/micro-benchmarks/DRB043-adi-parallel-no.c

[4]PolyOpt: http://web.cs.ucla.edu/~pouchet/software/polyopt

```
int hypre_ParCSRRelax_Cheby(....) {
...
   int num_rows = hypre_CSRMatrixNumRows(A_diag);...
   double *u_data
    = hypre_VectorData(hypre_ParVectorLocalVector(u));...
   double *orig_u;...
   orig_u = hypre_CTAlloc(double, num_rows);...
   double *ds_data, *tmp_data;...
   ds_data
    = hypre_VectorData(hypre_ParVectorLocalVector(ds));...
...
#ifdef HYPRE_USING_OPENMP
#pragma omp parallel for private(j) schedule(static)
#endif
      for ( j = 0; j < num_rows; j++ )
      {
         u_data[j] = orig_u[j] + ds_data[j]*u_data[j];
      }
...
}
```

Fig. 2: AMG2013 function `hypre_parCSRRelax_cheby` in file `parcsr_ls/par_relax_more.c`. The directive `omp parallel for` is at line 742 in the original file. This loop is verified to have no data race.

application AMG2013 is shown. We show only one verified parallel loop and some context from the source code. The function `hypre_ParCSRRelax_Cheby` takes a number of arguments and the arrays are allocated by called functions. An alias analysis has to ensure that the pointers `u_data` and `orig_u` and `ds_data` refer to different arrays. No pointer arithmetic occurs in the application and only pointers to the beginning of arrays are passed between functions. All offsets in arrays are accessed through index expressions. Once the property is established that the pointers do not alias and the memory regions referred to do not overlap, the polyhedral analysis can verify the property that no data race exists in this loop. The alias analysis is not integrated yet, we checked manually that no aliasing occurs, for details see Sec. IV-B.

III. Approach

DRACO performs data-race-free verification on a supported subset of OpenMP parallel loop nests in a program. It is implemented with the ROSE compiler infrastructure and takes advantage of the native support for OpenMP in ROSE, which parses OpenMP pragmas into special AST nodes and provides an analysis to identify whether a variable is shared among threads. DRACO also uses the Polyhedral Compiler Collection package (PoCC) [5] to perform the polyhedral data race analysis.

A. Precondition Checking

Not all parallel loops can be verified with the polyhedral model. The available support for OpenMP pragmas in DRACO and the assumptions made by the polyhedral model together result in the following preconditions that must be satisfied by a loop nest before it can be verified by the polyhedral data race analysis:

1) It is a for-loop nest parallelized by a supported OpenMP directive;
2) It contains no unsupported OpenMP directive or clause;
3) Its sequential version (i.e., without any OpenMP pragma) can be represented by the polyhedral model;
4) The arrays referenced in the loop nest do not overlap;
5) There is no out-of-bounds array access in the loop nest.

[5]PoCC: https://sourceforge.net/projects/pocc

The OpenMP directives currently supported by DRACO are `for`, `simd` and `for simd`. The supported clauses are `collapse` and data-sharing clauses. The handling of synchronization constructs is not supported and is subject of future work. The checking of OpenMP directives is implemented based on ROSE's OpenMP AST which contains OpenMP nodes representing corresponding pragmas.

The loop nests that can be represented and analyzed by the polyhedral model are called Static Control Parts (SCoP). In a SCoP the control flow and data dependencies can be computed statically. The branch conditions, loop bounds, and memory accesses in the loop nest are expressed as affine functions of induction variables and parameters. DRACO aims to parse loop nests that meet the first two preconditions into SCoPs and proceeds with the analysis only if the parsing is successful.

The last two preconditions are the assumptions made by the polyhedral analysis. We discuss the required alias analysis (Item 4) in Sec. IV-B. For bounds checking, we use a polyhedral-based method described in Sec. III-C.

B. Data Race Analysis

In a SCoP, two kinds of data race can exist. The first kind is the data race of induction variable references in loop initialization, test and increment expressions. This usually happens when the programmer forgets to declare the induction variables in loop nests as private. The second kind of data race is caused by non-induction variable references in the loop body, which is where the polyhedral analysis can be applied.

1) Induction Variable Data Race Analysis: In a SCoP, an induction variable is not written in the loop body. As a result, it causes a data race if and only if it is shared between threads. Thus the checking of data races among induction variables can be achieved by simply performing an analysis on the data-sharing clauses to identify shared variables. If there is a shared induction variable, a data race is found. Otherwise, there is no data race involving induction variables.

2) Loop Body Data Race Analysis: One key component of polyhedral loop optimizations is data dependence analysis, which is based on the polyhedral representation of statements in the loop body. A legal loop transformation needs to preserve all the data dependencies in the loop nest. In a parallelized loop nest, the violation of data dependencies of shared variables results in data races. DRACO employs the existing polyhedral data dependence analysis in PoCC to perform data-race-free verification on parallel loops nests.

Analysis-Enabling Program Transformation

The polyhedral dependence analysis in PoCC is not aware of OpenMP pragmas and thus does not distinguish between private and shared variable references. Since private memory accesses (by also considering pointer semantics) can never produce a data race, the dependence analysis should not take them into account. Instead of modifying PoCC's dependence

```
1   int main(int argc, char* argv[])
2   {
3     int i,j;
4     int n=100, m=100;
5     double b[n][m];
6   #pragma omp parallel for private(j)
7     for (i=1;i<n;i++)
8       for (j=0;j<m;j++)
9         b[i][j]=b[i][j-1];
10    printf ("b[50][50]=%f\n",b[50][50]);
11    return 0;
12  }
```

Fig. 3: DataRaceBench benchmark DRB014 with a data race caused by an out-of-bounds access in line 9.

analysis library, we remove the corresponding write references to private variables from the source code. We achieve this by making the left-hand side (a private variable reference) and the right-hand side of an assignment statement to two standalone statements, that replace the original statement. This transformation replaces private variable write references with read references and preserves all references to shared variables.

Polyhedral Dependence Analysis

After the previously mentioned code transformation, we apply a standard polyhedral dependence analysis described in [24]. The analysis is able to find all data dependencies between statement instance pairs. Since there is no write reference to private variables in the code, all the dependencies identified are caused by shared variable references.

Finding Data Races From Data Dependencies

We use the following conclusion drawn from the definitions of data races and data dependencies to identify data races from the result of data dependence analysis. For a loop nest L, let S_L denote the set of statements in the loop body of L. Let $D_{s_1 \to s_2}$ be a data dependency from s_1 to s_2. There is a data race between $s_1, s_2 \in S_L$ if and only if

$$\exists l \in \mathbb{Z}, \exists D_{s_1 \to s_2} \; carried \; at \; level \; l, s.t. \; 1 \leq l \leq C_L$$

where C_L is the collapse level of L decided by the `collapse` clause.

Following this conclusion, once all the data dependencies of shared variables are identified, a further filtering is needed to reach the final result. The filtering process takes the collapse level as input and keeps only the dependencies carried by loops whose nested level is less than or equal to the collapse level. If there is no dependency left, meaning that no data dependency is violated by the parallelization, the loop nest is data race free.

C. Polyhedral Bounds Checking via Transformation

For multi-dimensional arrays whose elements are stored contiguously in memory, an out-of-bounds access can be wrapped around and result in a data race within the same array. DataRaceBench contains such microbenchmarks with out-of-bounds array index operations that lead to data races (e.g. DRB014 as shown in Fig. 3). As previously mentioned in Sec. III-A, a precondition of the polyhedral analysis is that no out-of-bounds array access exists in the loop nest because

```
int a[10];

for (i = 0; i < 5; i++)
  for (j = 0; j < 5; j++)
    if (i >= j)
      a[i - j - 1]++;
```

(a) The original loop.

```
int a[10];
for (t = 0; t < 2; t++)
  for (i = 0; i < 5; i++)
    for (j = 0; j < 5; j++)
      if (i >= j)
        if (i - j - 1 < 0 ||
            i - j - 1 >= 10)
          tmp = 0;
```

(b) The transformed loop.

Fig. 4: A transformation example for bounds checking.

the wrap-around is equivalent to a modulo operation in the array index expression, making the expression non-affine.

The fulfillment of this precondition is necessary to the soundness of the data race freedom verification. For example, in Fig. 3, without the bounds checking that would otherwise stop us from further analyzing the loop, the polyhedral analysis will find no data dependency and miss the data race.

The checking of this precondition (i.e., bounds checking) can be done with the help of a polyhedral analysis when the size of each dimension of arrays is known at compile time.

Given a loop nest that is a SCoP, checking whether an array index expression can be out of bounds can be transformed into a dependence analysis problem and solved by the polyhedral analysis. We first build a temporary loop nest that has all the control structure containing the array reference in it. An if-statement that checks if the index expression is out of bounds is built at the location in the temporary loop nest that would be the array reference in the original loop nest. The branch of that if-statement has one temporary variable assignment (e.g., $tmp = 0$;). At last, a for-loop with two iterations is added to wrap the outermost loop of the temporary loop construction. Fig. 4 shows a simple example of the transformations. Once the temporary loop is built, we run polyhedral dependence analysis on it. If there is no dependency on a write to the temporary variable between different iterations, we can conclude that the index expression must be in-bounds.

This analysis can be overly conservative if there are parameters with values unknown at compile time in the polyhedral representation. DRACO only includes this analysis as an optional part of the pipeline and enables it when there is no parameter in the polyhedral representation. In our experiments we turned off the upper-bounds check because it turns out to be overly conservative. The lower-bounds check is performed.

D. Limitations

With all the preconditions mentioned in Sec. III-A satisfied, DRACO will produce no false negatives, which is necessary for the correctness of the data-race-free verification. But false positives can be present in the results. One kind of false positive is due to the existence of parameters whose values are unknown at compile-time. For example, consider a parameter that appears in a parallel loop's bound, whose value range limits that of the induction variable, and the loop can have a data race only if the induction variable falls into the impossible value range. Since the value range of the parameter is unknown during the polyhedral analysis, a false positive data race report could be issued. However, it is possible to perform a static pre-analysis to decide the possible values of a parameter so as to reduce the number of false positives.

Another kind of false positive is caused by the lack of support for some OpenMP clauses. An example is the `safelen` clause in the `simd` directive. This clause makes data races reported by DRACO between two array references with a distance larger than the safe length infeasible.

IV. Evaluation

We perform our evaluation with the data race benchmark suite DataRaceBench and the LLNL proxy application AMG2013. The input codes are unmodified, and we only replace the compiler in the Makefile with our analysis tool. Then the build process is run as usual except that it now generates race analysis results. Optionally the original compiler can be invoked after the analysis phase and the respective original file is also compiled (and linked) - this can be important for complicated build scripts, but turned out not to be relevant for our test cases.

A. Data Race Bench

In its initial release, version 1.0.1, DataRaceBench had 72 benchmarks. Further development with feedback and contributions from other research groups as well as the addition of microbenchmarks for addressing new features in OpenMP 4.5 have been made. The current release, version 1.2.0, contains 116 microbenchmarks. We used this latest version in our evaluation.

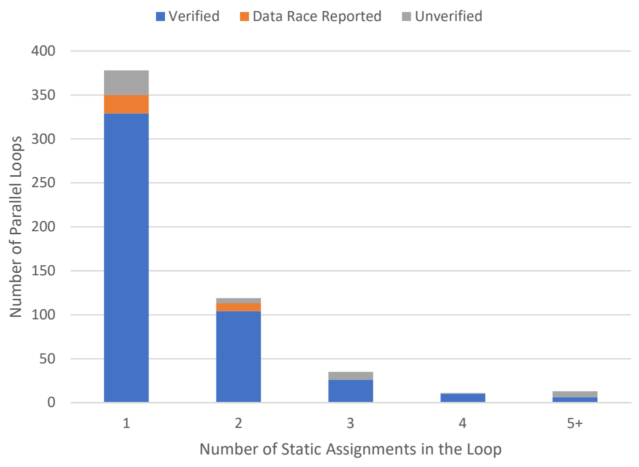

Fig. 5: DataRaceBench: verified parallel loops, loops with reported data races, and unverified parallel loops.

TABLE I: DataRaceBench result statistics

	Data Race Reported	Verified Data Race Free	Unverified	No Parallel Loop	Total
#benchmarks	26	20	42	28	116
Min. Runtime (s)	0.26	0.24	0.17	0.20	N/A
Max. Runtime (s)	0.63	1.32	208.44	7.17	N/A
Tot. Runtime (s)	12.95	7.83	233.46	25.16	279.40

```
! All parallel regions are parallel loops
* OpenMP parallel loop at line 6:
  Loop can be analyzed with the polyhedral model
  Number of data races: 1
  - Reason: flow dependency
    Source at line 8: a[2 * i + 1] = a[i] + 1;
    Sink at line 8: a[2 * i + 1] = a[i] + 1;
```

Fig. 6: DRACO's output on DataRaceBench DRB033.

Our prototype tool DRACO is run on each microbenchmark and reports one of the following results for each (possibly nested) parallel OpenMP loop:

- A potential data race is detected. The reason for the data race and the location information are reported.
- The loop is verified to be data race free and is guaranteed to be correct in this respect.
- The loop cannot be verified. The reason can be an unsupported OpenMP feature or because the loop is not an affine loop nest. Note that we also perform code transformations before the polyhedral analysis is performed which can make some loops analyzable for the polyhedral analysis (see Sec. III for more details).

Fig. 6 displays the output of DRACO after analyzing the DataRaceBench microbenchmark DRB033 shown in Fig. 1. The first line indicates that every parallel region in the input program is a parallel loop. This implies that the program can be fully verified if all its parallel loops are verified. The rest of the output gives the verification result of the only parallel loop starting from line 6 (the location of the `omp for` pragma). It identifies the data race caused by a flow dependency from the statement at line 8 to itself.

In Fig. 5, we show the verification results reported by the number of verified parallel loops across all 116 microbenchmarks. We show how many loops are verified to be data race free, in how many loop nests data races were reported, and how many loop nests remained unverified, grouped by the number of static assignments in the respective loop bodies. A static assignment is an assignment in the source code. There are a few benchmarks with a high number of parallel loops, e.g. DRB042 has 391 `omp simd` loops and several `omp for` loops. Only 4 benchmarks have more than 100 parallel loops.

Table I shows the results based on the number of benchmarks. A benchmark is reported as verified only if *all* parallel loops in the benchmark could be verified to be data race free. If a data race is detected in at least one parallel loop, the benchmark is reported as “Data Race Reported”. In all other cases, “Unverified” is reported. For example, in DRB013 there is a `nowait` loop causing a data race. We can verify the loop to be data race free, but not the entire region (similar DRB104). These benchmarks are reported as unverified. The column “No Parallel Loop” shows how many benchmarks do not contain any parallel loop.

Table I also shows initial runtime results for our prototype including the total time of analysis: parsing, pre-analysis to determine whether a loop can be analyzed with the polyhedral analysis, the polyhedral analysis itself (if applicable), and result reporting. It also shows the minimum, maximum and total analysis time, showing that for all benchmarks the analysis time is within reasonable time bounds. No analysis was aborted and each benchmark consists of one source file (= translation unit).

An interesting observation not captured in these tables is that the microbenchmarks contain 16 cases with variable-length arrays. The length of the array is a command line parameter to the benchmark. If no parameter is provided, some default value is selected. All 16 microbenchmarks contain a data race. For 13 of those 16 benchmarks in which we detected a data race, 3 were reported to be not verified.

We ran our experiments on a Linux machine running Ubuntu 16.04 with an Intel Core i7-8550U processor at 1.80 GHz and 16GB of RAM.

B. AMG2013 Proxy Application

AMG2013 is an LLNL proxy application that takes the BoomerAMG solver from Hypre and wraps it with a driver to run example problems. AMG2013 implements a parallel algebraic multigrid solver for linear systems arising from problems on unstructured grids. The driver provided in the benchmark can build various test problems. The default problem is a Laplace type problem on an unstructured domain with various jumps and an anisotropy in one part. AMG2013 is written in ISO-C. It is an SPMD code which uses MPI as well as OpenMP. MPI parallelism is achieved by domain decomposition. The driver provided with AMG2013 achieves this decomposition by subdividing the grid into logical P x Q x R (in 3D) chunks of equal size. The benchmark was designed to test parallel weak scaling efficiency. It consists of 75,000 lines of code. In our evaluation, we target the OpenMP parallel loops.

TABLE II: AMG2013 Loop Analysis Results

Data race reported	Verified Data Race Free	Unverified	Total	Runtime (s)
0	41	73	114	308.40

We ran DRACO on all 75,000 LOC, and analyzed each file separately. The analysis assumes that the arrays operated on by the polyhedral tool are alias-free because our prototype does not integrate whole-program analysis yet. We confirmed manually that there is no aliasing in the AMG2013 benchmark that violates the required assumptions. The same assumption of no aliasing is made in [3] and [6] where pointer analysis is considered as future work as well. AMG2013 has only some specific aliasing patterns: every array element is accessed through a pointer referring to the beginning of the array and an index offset. No pointer arithmetic is used. We plan to integrate the required alias analysis in DRACO to automate this aspect.

In Table II, we show the results of analyzing the loops in AMG2013. We can verify 41 of the 114 parallel OpenMP loops to be data race free (column 2) and no data race exists in the successfully analyzed loops (column 1). 73 loops remain unverified. The loops involve a number of different OpenMP directives and clauses. A more detailed view is presented in

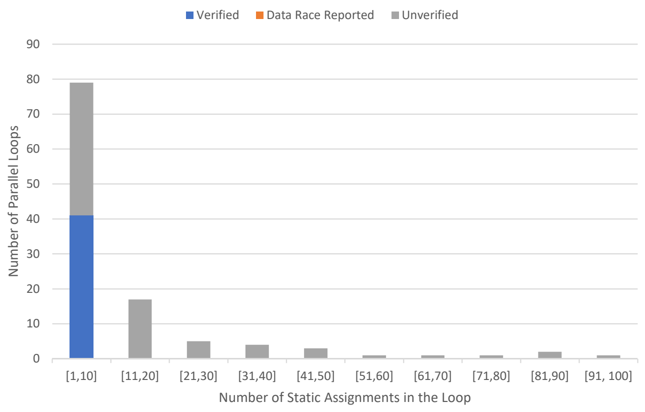

Fig. 7: AMG2013 Proxy Application: verified and unverified parallel loops. No loops with data races are reported.

Fig. 7. The histogram shows how many parallel loop nests could be verified and how many remain unverified for groups of loop nests with multiples of 10 assignments in the source code (static assignments) in the respective loop bodies. Those loops that cannot be verified are not affine loop nests (we also confirmed this manually).

We also analyzed AMG2013 with the tool Archer version 2.0.0. In our experiment Archer found three data races, matching the number of data races reported in a previous evaluation of Archer [2]. These three data races are inside an OpenMP parallel region but not in any parallel loop. Therefore, DRACO does not report them as data races.

V. Related Work

Data race detection has been a challenging problem for decades and has been attacked using dynamic and static analysis techniques. There exist also a number of approaches that combine several tools. We therefore discuss separately the related work for those three groups.

A. Static Data Race Detection

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 can find all data races. They might also report data races that do not exist due to the imprecision of static analysis. OmpVerify [3] is a static race detector that targets OpenMP exclusively using a polyhedral model to determine data dependencies in shared data. This approach is similar to our approach in terms of converting OpenMP programs into a polyhedral representation to enable data race detection. In [7] the polyhedral model is extended to represent SPMD programs and an approach for data race detection is presented for extracting race constraints that can be solved by an SMT solver such as Z3. In [6] this approach is further developed and implemented in the tool PolyOMP, support for pointer analysis is planned as future work. Those works assume that the preconditions of the polyhedral analysis are satisfied while our work takes the precondition checking into consideration. Unlike their methods that adapt the polyhedral analysis to parallel programs, our method transforms the source code to make use of the existing polyhedral analysis out-of-the-box. Besides, those works are only evaluated on a few small kernels, while we measured our method on a real-world application, showing the applicability of polyhedral methods on large-scale HPC applications.

Locksmith [19] is one such tool that seeks to correlate locks with the shared memory locations they guard. It over-approximates the set of data races, possibly returning some false positives. Another analysis seeks to improve the detection of shared variables [12] by performing pointer analysis in order to find global variables that are locally aliased.

CIVL [27] is a framework for static analysis and verification of concurrent programs. It supports MPI, OpenMP, CUDA, and Pthreads and can also perform a data race detection for OpenMP programs. It uses symbolic execution and can verify programs up to a certain bound, e.g. the number of loop iterations - in contrast to polyhedral analysis which allows verifying at least affine loop nests independent of the number of iterations. Due to its design of mapping the constructs of different parallel programming languages to a common intermediate representation, CIVL can also verify mixed programs, where, for example, both MPI and OpenMP are used.

B. Dynamic and Hybrid Data Race Detection

Dynamic data race detection tools run an instrumented target program and analyze the execution trace [26]. Many dynamic analyses use a happens-before approach. Reads and writes to shared memory are modeled by a partial order over events within the system [14]. This technique is heavily dependent on the application scheduler and may miss many latent races. 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 [8], [9], expanding it to single-threaded event-driven programs [16], defining additional relations such as casually-precedes [23], and by taking some high-level language semantics also in dynamic tools into account [20].

Lockset analyses such as Eraser [21] 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 [11], [17], [22]. These methods leverage information about local control flow, recent access, and common race patterns in order to dynamically adjust the analysis. This leads to greater flexibility when balancing accuracy and performance, as well as enabling long-term [26] and large-scale [22] analyses that might not be possible with other techniques.

In [15] the four dynamic analysis tools Helgrind, ThreadSanitizer, Archer, and Intel Inspector are evaluated using the DataRaceBench suite which provides benchmarks specifically designed for evaluating data race detection tools. There are groups of benchmarks with and without data races, and benchmarks that can be run with a command line parameter specifying the size of arrays that the program operates on. One of those four tools, Archer, uses also a polyhedral analysis to determine statically whether a loop-nest in a program is data race free. Archer is based on LLVM and leverages LLVM's polyhedral analysis for this purpose. Archer does not report in those cases the absence of data races, it only reports as usual that it did not detect a data race in the specific program run.

C. Related Analyses

Similar multi-tool analyses have been performed with other languages. Two targeting the Java language [1], [25] analyzed several data race detection tools and compared the accuracy and performance of each. The first [1] compared RaceFuzzer, RacerAJ, JCHORD, Race Condition Checker, and Java RaceFinder. The authors compared the compilation time, accuracy, precision, along with several other metrics. Java RaceFinder performed the best on their tests, although it only reported the first race found even if there were others in the program. In [25] the authors 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+ had the lowest overhead but missed at least one race that MultiLock found.

VI. Conclusion

In this paper, we presented an approach for proving the absence of data races using polyhedral analysis. Since polyhedral analysis can only handle affine loop nests the analysis cannot handle all loops in the analyzed application, but those loops that can be analyzed are verified for arbitrary loop bounds and iterations to be data race free. In addition, we presented a code transformation that widens the range of applicability of the polyhedral analysis (Sec. III-C). Using this approach we are able to determine that 40% of the microbenchmarks in DataRaceBench 1.2.0 are data race free or have a data race, and verify 41 of the 114 parallel loops of the proxy application AMG2013 are data race free. This is an important finding: 36% of the hand-written loops in AMG2013 are indeed affine loop nests and polyhedral analysis is suitable to verify that these loops are data race free.

In contrast to testing tools and bounded model checking tools, we can determine that a parallel OpenMP loop is data race free independent of a thread schedule and the number of iterations.

Polyhedral analysis allows us to determine there is no data race, but false positives are possible. The analysis may report a data race exists in a parallel loop when none is present. For DataRaceBench and the AMG2013 proxy application, no false positives are reported, but it is important to note that parallel loop patterns exist for which false positives can be reported. However, our analysis shows a high level of precision, as we do not report any false positives for the analyzed affine loops in DataRaceBench or AMG2013.

As part of our future work, we will integrate an inter-procedural alias analysis that will allow us to fully automate the verification of the evaluated programs.

If our analysis detects a data race, we report the line numbers of the conflicting accesses. As is common for verification tools, we plan to generate a counterexample that demonstrates how to trigger a data race. A counterexample has to involve at least two threads and the path that each of the threads can take in a program that eventually leads to a conflicting access. Counterexamples are also helpful in debugging and proving the existence of a data race.

On the path in achieving data-race-free programs, testing tools are helpful for detecting and debugging data races, but a verification tool is necessary to prove that a program is indeed data race free. Only then can a program (or at least parts of it) be considered correct and be trusted to behave as specified.

References

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

[2] S. Atzeni, G. Gopalakrishnan, Z. Rakamaric, D. H. Ahn, I. Laguna, M. Schulz, G. L. Lee, J. Protze, and M. S. Müller. ARCHER: effectively spotting data races in large openmp applications. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 53–62. IEEE, 2016.

[3] V. Basupalli, T. Yuki, S. Rajopadhye, A. Morvan, S. Derrien, P. Quinton, and D. Wonnacott. ompverify: Polyhedral analysis for the openmp programmer. In *Proceedings of the 7th International Conference on OpenMP in the Petascale Era*, IWOMP'11, pages 37–53. Springer-Verlag, 2011.

[4] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *Compiler Construction*, pages 283–303. Springer, 2010.

[5] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2008.

[6] P. Chatarasi, J. Shirako, M. Kong, and V. Sarkar. An extended polyhedral model for spmd programs and its use in static data race detection. In C. Ding, J. Criswell, and P. Wu, editors, *Languages and Compilers for Parallel Computing*, pages 106–120, Cham, 2017. Springer International Publishing.

[7] P. Chatarasi, J. Shirako, and V. Sarkar. Static data race detection for SPMD programs via an extended polyhedral representation. In *Proceedings of the Sixth International Workshop on Polyhedral Compilation Techniques, IMPACT*, volume 16, 2016.

[8] L. Effinger-Dean, B. Lucia, L. Ceze, D. Grossman, and H. Boehm. IFRit: interference-free regions for dynamic data-race detection. In G. T. Leavens and M. B. Dwyer, editors, *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 467–484. ACM, 2012.

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

[10] J. Holewinski, L.-N. Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *ICS*, June 2012.

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

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

[13] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan. When polyhedral transformations meet simd code generation. In *PLDI*, June 2013.

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

[15] 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*, SC '17, pages 11:1–11:14, New York, NY, USA, 2017. ACM.

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

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

[18] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong. Polyhedral-based data reuse optimization for configurable computing. In *FPGA*, Feb. 2013.

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

[20] J. Protze, M. Schulz, D. H. Ahn, and M. S. Müller. Thread-local concurrency: A technique to handle data race detection at programming model abstraction. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '18, pages 144–155, New York, NY, USA, 2018. ACM.

[21] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

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

[23] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan. Sound predictive race detection in polynomial time. In J. Field and M. Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 387–400. ACM, 2012.

[24] N. Vasilache, C. Bastoul, A. Cohen, and S. Girbal. Violated dependence analysis. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 335–344. ACM, 2006.

[25] M. Yu, S.-M. Park, I. Chun, and D.-H. Bae. Experimental performance comparison of dynamic data race detection techniques. *ETRI Journal*, 39(1):124–134, 02 2017.

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

[27] M. Zheng, M. S. Rogers, Z. Luo, M. B. Dwyer, and S. F. Siegel. CIVL: Formal verification of parallel programs. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 830–835, Nov 2015.

Appendix A
Artifact Description

A. *Benchmark and Proxy Application*

The data race benchmark suite and proxy applications used in this paper are open source and publicly available:

1) The DataRaceBench benchmark suite is available at https://github.com/LLNL/dataracebench.
2) The AMG2013 proxy application can be found at https://computation.llnl.gov/projects/co-design/amg2013.

B. *Evaluation with DRACO*

The tool DRACO was run on each DataRaceBench benchmark file and on each file of the proxy application AMG 2013. The used version of DRACO is 0.9.0.
For the evaluation on DataRaceBench, we used the following command line to analyze each microbenchmark:

```
draco --no-check-upper-bounds INPUT_FILE
```

With the option `--no-check-upper-bounds`, we turned off the additional checking of upper bounds of array accesses because this check is currently overly conservative. The polyhedral check of the lower array bounds remains active by default. We manually checked that no out of upper-bounds access exists. See section III-C for more details.
To analyze AMG2013, we modified the file Makefile.include of AMG2013 by adding the following line:

```
CC = draco --no-check-upper-bounds $(
    shell mpicc -show | cut -d' ' -f2-)
```

With the option `--compile`, DRACO can invoke the backend to compile the given input file. We did not use this option for the evaluation to avoid counting the compilation time as part of DRACO's runtime.

C. *Evaluation with Archer*

The tool Archer and installation instructions can be found at: https://github.com/PRUNERS/archer. The used version of Archer is 2.0.0 built on Clang/LLVM 6.0.1.
To instrument and build AMG2013 with Archer, we added the following line to the file Makefile.include of AMG2013:

```
CC = clang-archer --sa $(shell mpicc -
    show | cut -d' ' -f2-)
```

The command line parameters used in the evaluation are from an example provided in the AMG2013 documentation. The evaluation was run with the following command line parameters:

```
mpirun -np 1 ./amg2013 -laplace -n 40 40
    40 -P 1 1 1
```

The following list shows the data races as reported by Archer with file names, line numbers and column numbers:

1) AMG2013/utilities/threading.c:27:16
2) AMG2013/parcsr_ls/par_interp.c:1248:25
3) AMG2013/parcsr_ls/par_interp.c:1249:25

The data race analysis results reported by Archer are consistent with the results reported by our tool DRACO.