LLOV: A Fast Static Data-Race Checker for OpenMP Programs

UTPAL BORA, SANTANU DAS, PANKAJ KUKREJA, SAURABH JOSHI, and RAMAKRISHNA UPADRASTA, IIT Hyderabad, India
SANJAY RAJOPADHYE, Colorado State University, USA

In the era of Exascale computing, writing efficient parallel programs is indispensable, and, at the same time, writing sound parallel programs is very difficult. Specifying parallelism with frameworks such as OpenMP is relatively easy, but data races in these programs are an important source of bugs. In this article, we propose LLOV, a fast, lightweight, language agnostic, and static data race checker for OpenMP programs based on the LLVM compiler framework. We compare LLOV with other state-of-the-art data race checkers on a variety of well-established benchmarks. We show that the precision, accuracy, and the F1 score of LLOV is comparable to other checkers while being orders of magnitude faster. To the best of our knowledge, LLOV is the only tool among the state-of-the-art data race checkers that can verify a C/C++ or FORTRAN program to be data race free.

CCS Concepts: • **Software and its engineering → Compilers**; **Software testing and debugging**; Formal software verification; • **Computing methodologies → Shared memory algorithms**; **Parallel programming languages**;

Additional Key Words and Phrases: OpenMP, shared memory programming, static analysis, polyhedral compilation, program verification, data race detection

ACM Reference format:
Utpal Bora, Santanu Das, Pankaj Kukreja, Saurabh Joshi, Ramakrishna Upadrasta, and Sanjay Rajopadhye. 2020. LLOV: A Fast Static Data-Race Checker for OpenMP Programs. *ACM Trans. Archit. Code Optim.* 17, 4, Article 35 (November 2020), 26 pages.
https://doi.org/10.1145/3418597

1 INTRODUCTION

The benefits of heterogeneous parallel programming in obtaining high performance from modern complex hardware architectures are indisputable among the scientific community. Although indispensable for its efficiency, parallel programming is prone to errors. This is crucial, since programming errors could result in significant monetary losses or prove to be a risk factor where

This work is partially supported by a fellowship under Visvesvaraya PhD Scheme from MeitY, India (grant PhD-MLA/04(02)/2015-16), an Early Career Research award from SERB, DST, India (grant ECR/2017/001126), a Visvesvaraya Young Faculty Research Fellowship from MeitY (MeitY-PHD-1149), an NSM research grant (sanction number MeitY/R&D/HPC/2(1)/2014), a faculty research grant from AMD, and an Army Research Office grant (W911NF).
Authors' addresses: U. Bora, S. Das, P. Kukreja, S. Joshi, and R. Upadrasta, Dept. of CSE, IIT Hyderabad, Kandi, Telangana, 502285, India; emails: {cs14mtech11017, cs15mtech11018, cs15btech11029, sbjoshi, ramakrishna}@iith.ac.in; S. Rajopadhye, Dept. of CSE, Colorado State University, Fort Collins, Colorado, 80523, USA; email: Sanjay.Rajopadhye@colostate.edu.

1544-3566/2020/11-ART35
https://doi.org/10.1145/3418597

human safety systems are involved. Historical incidents such as Therac-25 accidents [45], the Ariane 5 flight 501 failure [55], EDS Child Support IT failure, and Knight Capital Group trading glitch [74] have been directly attributed to software errors and testify to the need for bug detection mechanisms. The detection of errors a priori, therefore, could significantly reduce this risk and make programs more robust and dependable.

In this article, we propose a solution to the problem of statically detecting data race errors in OpenMP parallel programs. We developed a data race detection tool based on LLVM/Clang/Flang [58, 60, 61] that is amenable to various languages, such as C, C++ as well as FORTRAN. To the best of our knowledge, our work is the *first static OpenMP data race detection tool based on the language independent intermediate representation of LLVM* (henceforth called LLVM-IR) [56]. Specifically, we make the following contributions:

- Implementation of a *fast*, *static*, and *language agnostic* OpenMP data race checker in the LLVM framework based on its intermediate representation (LLVM-IR) using the polyhedral framework Polly [36]. Our tool can also certify that a program is *data race free* along with detecting data races in OpenMP programs. Moreover, our tool provides a limited support for non-affine programs using Mod/Ref information from the Alias Analyzer of LLVM. Additionally, the tool can be used to generate and visualize the task graph (exportable as a file in dot/gv format) of an OpenMP program.
- We create *DataRaceBench FORTRAN*, a FORTRAN manifestation of DataRaceBench v1.2 [51], and release it under open source [43]. The latter is a benchmark suite consisting of programs written in C/C++ using OpenMP kernels. Our DataRaceBench FORTRAN allows for standardized evaluation of tools that can analyze FORTRAN programs for data races.
- We make a comparative study of well-known data race checker tools on a standard set of OpenMP benchmarks. We evaluate these tools on various metrics such as precision, recall, accuracy, F1 score, Diagnostic Odds Ratio, and running times. We show that LLOV performs quite well on these metrics while completely outperforming its competitors in terms of runtime.

The rest of the article is organized as follows: We start with the motivation for our work in Section 2 and describe common data races in Section 3. In Section 4, we discuss the verifier implementation details and our proposed algorithm and list out the advantages of our approach over the existing dynamic tools. Section 5 discusses related work in OpenMP data race detection along with their differences from our approach. Our results and comparison with other verifiers are presented in Section 6, and, finally, we conclude in Section 7.

2 BACKGROUND AND MOTIVATION

Multithreading support in hardware architectures has been very common in recent times with the number of cores per socket going up to 56 in Intel Xeon Platinum 9282 with 2 threads per core and up to 72 in Intel Xeon Phi Processor 7290F (accelerator) with 4 threads per core. The Top500 [85] November 2018 list of supercomputers comprises systems with cores per socket ranging from 6 to 260. As these are simultaneous multithreading (SMT) systems, operating systems with support for SMT and/or Symmetric Multi Processing (SMP) can benefit from execution of a large number of threads in parallel. The memory (not cache) is shared among all the threads in a node with either uniform or non-uniform memory access (UMA/NUMA), enabling shared memory multithreading.

In the past, the scientific community wrote parallel programs in C/C++ and FORTRAN using either language extensions or with APIs to run them across different nodes in a cluster or grid. With the advent of multi-core processors, the focus shifted to a shared memory programming

model, e.g., the `pthreads` library/runtime system [75] coupled with a vanilla language like C/C++, or a parallel language like coarray-FORTRAN or HPF.

In recent years, languages having structured parallelism such as Cilk [14], Julia [12], Chappel [18, 23], X10 [19], and others started gaining popularity in the community. However, the community has extensively adopted structured parallel programming frameworks, such as OpenMP [24, 69], MPI [64], OpenACC [68], and OpenCL [38], because of easy migration from legacy sequential code. The availability of efficient runtime systems and versatile support for various architectures played a major role in popularizing these frameworks. Amongst these, in this work we focus on the OpenMP parallel programming framework.

The OpenMP programming paradigm [24, 69] introduced structured parallelism in C/C++ and FORTRAN. It supports a *Single Program Multiple Data* (SMPD) programming model with multiple threads, a *Single Instruction Multiple Data* (SIMD) programming model within a single thread in CPUs with SIMD hardware extension, as well as SIMD among threads of a *thread block* in GPUs. OpenMP enables divide-and-conquer paradigm with tasks and nested parallelism, provides a data environment for shared memory consistency, and supports mutual exclusion and atomicity and synchronization amongst threads.

However, incorrect usage of OpenMP may introduce bugs into an application. A common data access anomaly, referred to as data race, occurs where two threads incorrectly access the same memory location and is defined formally as follows.

Definition 2.1 (Data Race). An execution of a concurrent program is said to have a *data race* when two different threads access the same memory location, these accesses are not protected by a mutual exclusion mechanism (e.g., locks), the order of the two accesses is non-deterministic and one of these accesses is a write.

Though compilers do ensure that OpenMP constructs conform to the syntactic and semantic specifications [70], *none of the mainstream compilers*, such as GCC [35], LLVM [58], and PGI [72], provide *built-in data race detection* support. There exist dynamic tools to detect data races, but they either take a very long time to report races or might miss some data races. This is because these tools are dependent on the execution schedule of the threads and the program parameters. The primary goal of our work is to provide a built-in data race detector for OpenMP parallel programs in the LLVM toolchain, using static analysis technique as discussed in Section 4.

Definition 2.2 (Team of threads). A set of OpenMP threads comprising a master thread and an optional group of sibling threads that participate in the potential execution of an OpenMP parallel region is called a *team*. The master thread is assigned thread id zero.

By default, OpenMP considers variables as *shared* among all the threads in a team.

3 COMMON DATA RACES IN OPENMP PROGRAMS

In this section, we will walk through, with examples, different data races frequently encountered in OpenMP programs. Note that we are expansive in setting the stage here, not all the races described below can be detected by LLOV.

3.1 Missing Data Sharing Clauses

Listing 1 shows an OpenMP worksharing construct `omp parallel for` with a data race. The program computes the sum of squares of all the elements in the matrix `u`. Here, variables `temp`, `i`, and `j` are marked as `private`, indicating that each thread will have its own copy of these variables. However, the variable `sum` (Line 5) of Listing 1 is not listed by any of the data sharing clauses. Therefore, the variable `sum` will be *shared* among all the threads in the team. Thus, each thread

```
1 #pragma omp parallel for private (temp,i,j)
2   for (i = 0; i < len; i++)
3     for (j = 0; j < len; j++) {
4       temp = u[i][j];
5       sum = sum + temp * temp;
6     }
```

Listing 1. DRB021: OpenMP Worksharing construct with data race.

```
1  !$OMP PARALLEL DO
2     do i = 0, len - 1
3          tmp = a(i) + i
4          a(i) = tmp
5     end do
6  !$OMP end PARALLEL do
```

Listing 2. DRBF028: FORTRAN code with data race because of missing private clause.

```
1 for (i=0;i<n;i++) {
2 #pragma omp parallel for
3   for (j=1;j<m;j++) {
4     b[i][j]=b[i][j-1];
5   }
6 }
```

Listing 3. DRB038: Example with Loop Carried Dependence.

```
1 for (k = 1; k <= n; k++)
2   #pragma omp parallel for
3   for (i = 1; i <= n; i++)
4     for (j = 1; j <= n; j++)
5       A[i][j] =
6         min(A[i][k] + A[k][j], A[i][j]);
```

Listing 4. Parallel Floyd-Warshall Algorithm with a benign race condition.

will work on the same shared variable and update it simultaneously without any synchronization, leading to a data race.

Listing 2 presents a program in FORTRAN with a data race due to a missing `private` clause corresponding to the variable `tmp` (Line 3). Due to such intricacies, a programmer is prone to make mistakes and inadvertently introduce data races in the program.

Our aim is to develop techniques and a tool that understand the semantics of OpenMP pragmas and clauses with all their subtleties.

3.2 Loop Carried Dependencies

OpenMP programs may suffer from data races due to incorrect parallelization strategies. Such data races may occur because of parallelization of loops with loop carried dependencies.

For example, the loop nest in Listing 3 is parallel in the outer dimension (Line 1), but it is the inner loop (Line 3) that is marked parallel, which has a loop carried dependence, because the read of b[i][j-1] (Line 4) is dependent on write to b[i][j] (Line 4) in the previous iteration. A correct parallelization strategy for this example would be to mark the outer loop as parallel in place of the inner loop.

As another example, Listing 4 is a parallel implementation of Floyd-Warshall's shortest path algorithm with a *benign*[1] race condition [8]. This is because all the iterations of the (parallel) inner j loop read A[i][k] including the one where j=k, which also writes into A[i][k]. Hence, the iterations before j=k need the previous value of A[i][k] and subsequent ones need the new, updated value. However, if all the matrix entries are non-negative, then A[i][k] is unchanged by the assignment, and therefore the race introduced by parallelizing the j loop is benign.

3.3 SIMD Races

OpenMP supports SIMD constructs for both CPUs and GPUs. In CPUs, SIMD is supported with vector processing units, where the consecutive iterations of a loop can be executed in a SIMD

[1]A race is said to be benign (see Reference [65] for examples and analyses) if it can be formally proved that the result of the computation is unaffected by it.

```
1 #pragma omp simd
2 for (int i=0; i<len-1; i++){
3   a[i+1] = a[i] + b[i];
4 }
```

Listing 5. DRB024: Example with SIMD data race.

```
1 #pragma omp parallel shared(b, error) {
2 #pragma omp for nowait
3     for(i = 0; i < len; i++)
4       a[i] = b + a[i]*5;
5 #pragma omp single
6     error = a[9] + 1;
7   }
```

Listing 6. DRB013: Example with data race due to improper synchronization.

processing unit within a single core by a single thread. This is contrary to other loop constructs where iterations are shared among different threads.

In the example in Listing 5, the loop is marked as SIMD parallel loop by the pragma `omp simd` (Line 1). This signifies that consecutive iterations assigned to a single thread can be executed concurrently in SIMD units, called vector arithmetic logic units (ALU), within a single core rather than executing sequentially in a scalar ALU. However, because of the forward loop carried dependence between write to a[i+1] (Line 3) in one iteration and read of a[i] (Line 3) in the previous iteration, concurrent execution of the consecutive iterations in a vector ALU will produce inconsistent results. Dynamic data race detection tools fail to detect data races in such cases as the execution happens within a single thread.

3.4 Synchronization Issues

Improper synchronization between threads is a common cause of data races in concurrent programs. OpenMP can have both explicit and implicit synchronizations associated with different constructs.

The constructs `parallel`, `for`, `workshare`, `sections`, and `single` have an implicit barrier at the end of the construct. This ensures that all threads in the team wait for others to proceed further. This enforcement can be overcome with the `nowait` clause where threads in a team, after completion of the construct, are no longer bound to wait for the other unfinished threads. However, improper use of the `nowait` clause can result in data races as shown in Listing 6. In this example, a thread executing the `parallel for` (Line 3) will not wait for the other threads in the team because of the `nowait` clause (Line 2). Threads that have finished executing the `for` loop are free to continue and execute the `single` construct (Line 5). Since there is a data dependence between write to a[i] (Line 4) and read of a[9] (Line 6), it might result in a data race.

Such cases are extremely difficult to reproduce as they are dependent on the order of execution of the threads. This particular order of execution may not manifest during runtime, therefore, making it hard for dynamic analysis tools to detect such cases. Static analysis techniques have an advantage over the dynamic techniques in detecting race conditions for such cases.

3.5 Control Flow Dependent on Number of Threads

Control flow dependent on number of threads available at runtime might introduce data races in a parallel program. In the example in Listing 7, data races will arise only when thread IDs of two or more threads in the team are multiples of 2.

4 IMPLEMENTATION DETAILS

In this section, we describe the architecture, the implementation and the algorithm of our tool.

LLOV is built on top of LLVM-IR and can analyze OpenMP programs written in C/C++ or FORTRAN. In principle, any programming language that has a stable LLVM frontend can be supported.

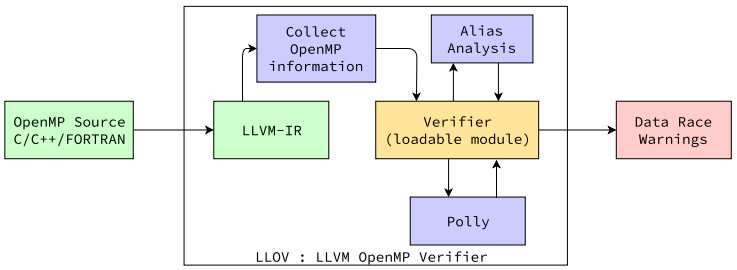


Fig. 1. Flow diagram of LLVM OpenMP Verifier (LLOV).

```
#pragma omp parallel
  if (omp_get_thread_num() % 2 == 0) {
    Flag = true;
  }
```

Listing 7. Control flow dependent on number of threads.

LLVM-IR can be generated from C/C++ programs using the Clang [60] frontend and from FORTRAN programs using the Flang [61] frontend. The architecture of LLOV is shown in Figure 1. The LLOV algorithm is primarily based on the one used in ompVerify [8].

We cover the architecture and implementation details in Section 4.1, followed by the algorithm of our tool in Section 4.2. We discuss the advantages of our tool over dynamic race detection tools in Section 4.3, and, finally, we list the limitation of the current version of our tool in Section 4.4.

4.1 LLOV Architecture

LLOV has two phases: analysis and verification.

4.1.1 Analysis Phase. In the first phase, we analyze the LLVM-IR to collect various OpenMP pragmas and additional information required for race detection. This analysis is necessary, because OpenMP constructs are lowered to the IR by compiler frontends such as Clang [60] and Flang [61]. LLVM-IR [56] is sequential and does not have support for parallel constructs. Hence, parallel constructs in high-level languages are represented in the IR as function calls. OpenMP pragmas are translated as function calls to the APIs of the OpenMP runtime library *libomp*.

As part of this analysis, for each OpenMP construct, we collect information such as memory locations, access types, storage modifiers, synchronization details, scheduling type, and so on. An *in-memory representation* of the OpenMP directive contains a directive type, a schedule (if present), variable names and types, and a list of child directives for nested OpenMP constructs. An illustration of our representation is shown in Listing 8 for the example in Listing 6 (Section 3). The grammar for this representation is presented in BNF form in Listing 9.

Reconstructing the OpenMP information from the IR has many challenges. Not all pragmas are handled directly by the runtime. Directives for SIMD constructs are just a hint to the optimizer that the program segment could be executed in parallel by SIMD units. Some constructs such as worksharing `for` and `sections` are very similar once they are translated to LLVM-IR. It becomes a challenge to distinguish between the two. Recovering worksharing `for` becomes challenging when the `collapse` clause is used. The resulting IR has no information about the loop nest in the original program. The data sharing clauses such as `private`, `shared`, `firstprivate`, and `lastprivate` are not explicitly annotated in the IR. They require additional analysis and we reconstruct them using their properties. Only `reduction` and `threadprivate` variables can be extracted easily. We

```
Directive:  OMP_Parallel
  Variables:
    Private:   %.omp.ub = alloca i32, align 4
    Private:   %.omp.lb = alloca i32, align 4
    Shared: i32* %i
    Shared: i32* %len
    Firstprivate: i64 %vla
    Shared: i32* %a
    Shared: i32* %b
    Shared: i32* %error
    Private:   %.omp.stride = alloca i32, align 4
    Private:   %.omp.is_last = alloca i32, align 4
  Child Directives:
  1:  Directive:    OMP_Workshare_Loop
    Schedule type : Static Schedule (auto-chunked)
  2:  Directive:    OMP_Workshare_single
  3:  Directive:    OMP_Barrier
```

Listing 8. In-memory representation of a directive.

```
<Directive> ::= <Dtype> [ Sched ]
       { <Var> } { <Directive> }
<Dtype> ::= parallel | for | simd
        | workshare | single
        | master | critical
<Var> ::= <Vtype> val
<Vtype> ::= private | firstprivate
        | shared | lastprivate
        | reduction | threadprivate
<Sched> ::= [ <modifier> ]
            [ ordered ] <Stype>
            <chunk>
<modifier> ::= monotonic
        | nonmonotonic
<Stype> ::= static | dynamic
        | guided | auto | runtime
<chunk> ::= positive-int-const
```

Listing 9. BNF of in-memory representation of directives.

Table 1. Comparison of OpenMP Pragma Handling by OpenMP-aware Tools

OpenMP Pragma	LLOV	ompVerify	PolyOMP	DRACO	SWORD	Archer	ROMP
#pragma omp parallel	Y	Y	Y	Y	Y	Y	Y
#pragma omp for	Y	Y	Y	Y	Y	Y	Y
#pragma omp parallel for	Y	Y	Y	Y	Y	Y	Y
#pragma omp critical	N	N	N	N	Y	Y	Y
#pragma omp atomic	N	N	N	N	Y	Y	Y
#pragma omp master	N	N	Y	N	Y	Y	Y
#pragma omp single	N	N	Y	N	Y	Y	Y
#pragma omp simd	Y	N	N	Y	N	N	N
#pragma omp parallel for simd	Y	N	N	Y	N	N	N
#pragma omp parallel sections	N	N	N	N	Y	Y	Y
#pragma omp sections	N	N	N	N	Y	Y	Y
#pragma omp threadprivate	Y	N	N	N	N	Y	Y
#pragma omp ordered	Y	N	N	N	N	Y	Y
#pragma omp distribute	Y	N	N	N	N	Y	Y
#pragma omp task	N	N	N	N	N	Y	Y
#pragma omp taskgroup	N	N	N	N	N	Y	Y
#pragma omp taskloop	N	N	N	N	N	Y	Y
#pragma omp taskwait	N	N	N	N	N	Y	Y
#pragma omp barrier	N	N	Y	N	Y	Y	Y
#pragma omp teams	N	N	N	N	N	N	N
#pragma omp target	N	N	N	N	N	N	N
#pragma omp target map	N	N	N	N	N	N	N

Y for Yes, N for No.

overcome the challenge to extract OpenMP pragmas from IR by conservatively recognizing patterns in IR that are generated from high level source code that use these pragmas.

4.1.2 Verification Phase. LLOV checks for data races only in regions of a program marked parallel by one of the structured parallelism constructs of OpenMP listed in Table 1. A crucial property of OpenMP constructs is that its specification [70] allows only *structured blocks* within a pragma. A structured block must not contain arbitrary jumps into or out of it. In other words, a structured block closely resembles a Single Entry Single Exit (SESE) region [2] used in loop analyses. To our advantage, the polyhedral framework Polly [36] also builds SESE regions before applying its powerful and exact dependence analysis and complex transformations.

The polyhedral framework is based on *exact dependence (affine) analysis* [33], by which the dependence information can be expressed as (piecewise, pseudo-) affine functions. Polly [36, 62] is the polyhedral compilation engine in LLVM framework. Polly relies on the Integer Set Library (ISL) [89] to perform exact dependence analysis, using which Polly performs transformations such

as loop tiling, loop fusion, and outer loop vectorization. Dependencies are modelled as ISL relations and transformations are performed on integer sets using ISL operations. The input to Polly is serial code that is marked as Static Control Part (SCoP) and the output is tiled or parallel code enabling vectorization.

LLOV is built using the Polly infrastructure but does not use its transformation capabilities. Our primary goal is to perform analyses, whereas Polly is designed for complex parallelizing or locality transformations followed by polyhedral code generation. The input to LLOV is *explicitly parallel code* that uses the structured parallelism of OpenMP. With an assumption that the input code has a serial schedule,[2] LLOV calculates its dependence information by using the dependence analyzer of Polly. Using this dependence information, LLOV then analyzes the parallel constructs and checks the presence or absence of data races. Consequently, LLOV can deterministically state whether a program has data races or whether it is race free for an affine subset of programs.

Polly was designed [36, 37] as an automatic parallelization pass using polyhedral dependence analysis. Its analysis and transformation phases were closely coupled and the analyses were not directly usable from outside Polly, neither by other analyses nor by optimization passes in LLVM. In particular, SCoP detection and dependence analysis of Polly was not directly accessible from analyses in LLVM.

We modified and extended the analysis phase of Polly in such a way that its internal data structures become accessible from LLVM.[3] Other changes involved modifying the dependence analysis of Polly so that its Reduced Dependence Graph (RDG) could be computed *on-the-fly* for a function, thereby reducing the analysis time for LLOV. Polly can detect and model only sequential programs. Hence, it does not support OpenMP programs. We incorporated changes to the SCoP detection to model OpenMP programs assuming a sequential schedule. The OpenMP parallel LLVM-IR contains runtime library calls to change the loop bounds at runtime, which makes the program non-affine. We mitigate this problem by resetting the loop bounds to the original values.

The two phases of LLOV are not tightly coupled, meaning the verification phase is separate from the analysis phase. The advantage of having a two phase design is that the verifier could easily be plugged with another analysis phase for other parallel programming APIs such as Intel TBB [41, 73], OpenCL [38] once they have a translator to LLVM-IR.

4.2 Race Detection Algorithm

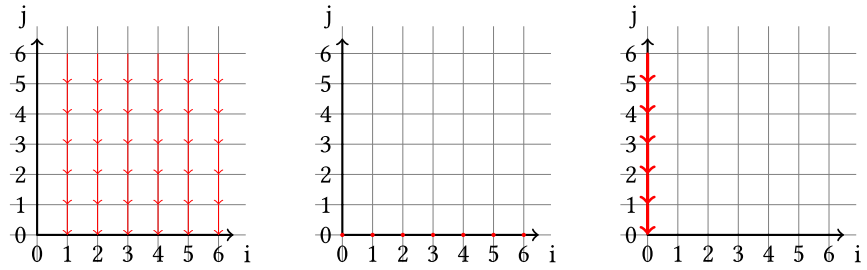
First, we cover the race detection for affine regions that relies on Polly, followed by the race detection for non-affine regions that relies on the Alias Analysis of LLVM.

4.2.1 Race Detection in Affine Regions. In the analysis phase, we gather information provided by OpenMP's structured parallelism constructs. We model a section of code, marked as parallel by one of the OpenMP constructs listed in Table 1, as static affine control parts (SCoPs) in the polyhedral framework. Our race detection algorithm runs only on sections of a program marked as parallel by one of these pragmas. This reduces analysis time of LLOV as it can avoid performing dependence analysis on the sequential fragments of the program.

For each SCoP, we query the race detection Algorithm 1 to check for the presence of dependences. A data race is flagged when the set of the memory accesses in the reduced dependence graph (RDG) and the set of the shared memory accesses within the SCoP is not disjoint. When dependencies are absent, the SCoP is parallel and the corresponding program segment is guaranteed to be data race free. Hence, we can verify—fully statically—the *absence of data race* in a program.

[2]This assumption is trivial to prove because of the input C language semantics.

[3]The initial version of the implementation for exposing Polly's dependence analysis information to LLVM was published in Polly (as part of *Google Summer of Code* 2016 project) *"Polly as an analysis pass in LLVM"* [17].



(a) Dependence Polyhedra (b) Projection of 2a on idimension (c) Projection of 2a on Jdimension

Fig. 2. Dependence polyhedra and its projections on i and j dimensions.

```
for (i=0;i<m;i++) {
  for (j=1;j<n;j++) {
    b[i][j]=b[i][j-1];
  }
}
```

Listing 10. Two-dimensional loop nest.

ALGORITHM 1: Race Detection Algorithm

Input: Loop L
Output: True/False
Function isRaceFree(L):
  $SCoP$ = ConstructSCoP(L) ;
  RDG = ComputeDependences($SCoP$) ;
  $depth$ = GetLoopDepth(L) ;
  if *isParallel(RDG, depth)* **then**
    // Program is race free.
    return True ;
  else
    // Data Race detected.
    return False ;
  return result
End Function

ALGORITHM 2: Algorithm to check parallelism

Input: RDG, Loop-depth dim
Output: True/False
Function isParallel(RDG, dim):
  if *RDG is Empty* **then**
    return True ;
  else
    Flag = True;
    while *Dependence D in RDG* **do**
      D' = Project Out all dimensions except dim from D ;
      if *D' is Empty* **then**
        continue ;
      else
        Flag = False ;
        break ;
    return Flag ;
End Function

The polyhedral representation of the affine static control program in Listing 10 consists of an iteration domain (**I**), an execution order called schedule (**S**), and an access function (**A**) mapping iteration number to memory accesses. The RDG (**D**) is computed using this information. The RDG is shown graphically in Figure 2(a).

Iteration Domain : $\mathbf{I} = \{\mathtt{S0}(i,j) : 0 \leq i \leq m-1 \wedge 1 \leq j \leq n-1\}$
Schedule : $\mathbf{S} = \{\mathtt{S0}(i,j) \rightarrow (i,j)\} \cap_{dom} \mathbf{I}$
Access Map : $\mathbf{A} = \{\mathtt{S0}(i,j) \rightarrow \mathtt{M}(i,j); \mathtt{S0}(i,j) \rightarrow \mathtt{M}(i,j-1)\}$
Dependencies : $\mathbf{D} = \{\mathtt{S0}(i,j) \rightarrow (i,j-1) : 0 \leq i \leq m-1 \wedge 1 \leq j \leq n-1\}$

Figure 2(b) shows the projection of the RDG on i-dimension, which results in vectors with zero magnitude as represented by red dots. This signifies that the loop is parallel in the i-dimension. Figure 2(c) shows the projection of the RDG on j-dimension, which are vectors of unit magnitude. Non-zero magnitude means that this dimension is not parallel.

Time complexity of the race detection algorithm is exponential in the number of inequalities present, since our approach is based on Fourier–Motzkin elimination.

4.2.2 Race Detection in Non-affine Regions. In addition to race detection in affine regions that can be exactly modelled by Polly, we use LLVM's Alias Analysis (AA) to conservatively analyze non-affine regions that cannot be modelled by Polly.

We use the Mod/Ref information from the Alias Analysis engine of LLVM [59] to analyze whether a shared memory location is read (Ref) or modified (Mod) by an instruction. The AA engine provides generic helper functions to return the Mod/Ref information for a memory location and an instruction of one of the following types: callsite, load, store, atomic read-write, invoke, and so on. The Mod and Ref bits are set for an instruction if the execution of the instruction might modify or reference the specified memory location.

If an operation inside a parallel region on the specified memory location is not protected by locks and Mod/Ref is set, then LLOV flags a race signaling a potential data race. The AA race checks are invoked only when the region cannot be modelled by Polly as an affine region. The Mod/Ref analysis of LLVM is conservative, and can lead to LLOV producing false-positive races. Thus LLOV provides a limited support of non-affine programs.

These AA-based checks for race detection in LLOV are enabled by default; we also provide a flag (*-openmp-verify-disable-aa*) that can be used to disable these and run only the polyhedral verifier of LLOV.

4.3 Advantages over Dynamic Race Detection Tools

Our static race detection tool LLOV has several advantages over state-of-the-art dynamic race detection tools.

4.3.1 Detects Races in SIMD Constructs. LLOV can detect SIMD races within a single thread, such as those shown in Listing 5. It can detect parallelism of the SIMD loop within the loop nest, but even when the loop is not parallel, there is a possibility of data races due to concurrent execution of consecutive iterations in the SIMD units within a single core. Dynamic race detection tools are based on different techniques such as vector clocks, happens-before relations, locksets, monitors, offset-span-labels, and so on, and fail to detect such race conditions present within a single thread.

4.3.2 Independent of Runtime Thread Schedule. Being a static analysis tool, LLOV has the added advantage that the race detection is not dependent on the order of the execution of the threads. This is a major drawback of the dynamic tools, because they need to be run multiple times for each specific number of threads to detect races dependent on the execution order.

4.3.3 Independent of Input Size. LLOV can handle parametric array sizes and loop bounds. Since we solve the problem with parametric integer programming [32], there is no limitation on the input size, provided the control flow is not affected by it. However, the dynamic data race detection tools need to be run multiple times for each program parameter to capture races. It is computationally not feasible to cover all the possible input sizes and hence, a dynamic tool can never be complete.

4.3.4 Independent of Number of Threads. Our analysis is not dependent on the number of threads available during runtime. However, all known dynamic tools have to be run multiple times

with different numbers of threads to detect races. Hence, dynamic tools might miss out race conditions when the number of runtime threads is small.

4.4 Limitations of LLOV

LLOV is in active development and in the current version, we attempted to cover the frequently used OpenMP v4.5 pragmas. In the current version, LLOV does not provide support for the OpenMP constructs for synchronization, device offloading, and tasking. Function calls within an OpenMP construct are handled by LLOV only if the function is inlined. Also, since our tool is primarily based on the polyhedral framework, its application is limited by the affine restrictions. However, LLOV provides a limited support for non-affine programs using Mod/Ref analysis as discussed in Section 4.2.2. Programs with dynamic control flow and irregular accesses (like $a[b[i]]$) fall outside the purview of the polyhedral framework. We are working on extending the analysis of our tool on such non-affine programs, but that is beyond the current scope.

LLOV may produce False Negatives when there is a race due to a dependence across two static control parts (SCoPs) of a program. One such category of programs is the presence of `nowait` clause in a worksharing `for` construct.

LLOV can produce False Positives for programs with explicit synchronizations with barriers and locks. Programs generated by automatic parallelization tools such as PolyOpt [76] and PLuTo [15] will have complex loop bounds and are difficut to model precisely. LLOV might produce FP for some of these automatically generated tiled or parallel kernels.

Handling `sections` construct in LLVM-IR is a challenge, as the resulting IR is similar to worksharing `for` construct but with a control dependence on the number of threads. Hence LLOV might produce FP/FN cases instead of showing a diagnostic message stating that the construct is outside its purview.

5 RELATED WORK

There has been extensive work on data race detection in parallel programs; many static, dynamic, and hybrid analyses approaches have been proposed. Mellor-Crummey et al. [63] proposed race detection in fork-join multithreaded programs using Offest-Span labelling of the nodes. Eraser [80] proposed lockset-based approach for race detection. Most of the earlier works have focused on `pthread`-based programs, while recent works such as ompVerify [8], Archer [4], ROMP [39], SWORD [5], DRACO [91], and PolyOMP [20] have targeted OpenMP programs.

In the following subsections, we discuss the state-of-the-art tools and categorize them based on their analyses and their approaches.

5.1 Static Tools

There are multiple static data race detectors in the literature. Common techniques used for static analyses are lockset-based approach or modeling data races as a linear programming problem (integer-linear or parametric integer-linear) and appropriately using an ILP or SMT solver to check for its satisfiability. Here, we briefly cover the state-of-the-art static data race detection tools for OpenMP programs. We limit the discussion to OpenMP race detection tools only.

ompVerify [8] is a polyhedral model-based static data race detection tool that detects incorrectly specified `omp parallel for` constructs. ompVerify computes the Polyhedral Reduced Dependency Graph (PRDG) to reason about possible violations of true dependencies, write-write conflicts, and stale read problems in the OpenMP `parallel for` construct. Although our approach is inspired by ompVerify, we have implemented a different algorithm to detect parallelism of loop nests. And, while ompVerify can handle only the `omp parallel for` construct, the coverage of LLOV is *much wider*; it can handle many more pragmas as listed in Table 3. Moreover,

the prototype implementation of ompVerify is in Eclipse CDT/CODAN framework using the AlphaZ [93] polyhedral framework, whereas LLOV is based on the widely used LLVM compiler infrastructure. Finally, ompVerify works at the AST level, whereas LLOV works on the language independent LLVM-IR level making it applicable to multiple languages. LLOV also has a limited support for non-affine regions that ompVerify does not have.

DRACO [91] is a static data race detection tool based on the Polyhedral model and is built on the ROSE compiler framework [79, 81]. One significant advantage of LLOV over DRACO is that LLOV is based on LLVM-IR and is language agnostic, while DRACO is limited only to the C family of languages that can be compiled by the ROSE compiler.

PolyOMP [20, 21] is a static data race detection tool based on the polyhedral model. PolyOMP [21] uses an extended polyhedral model to encode OpenMP loop nest information as constraints and uses the Z3 [25] solver to detect data races. The extended version of PolyOMP [20] uses May-Happen-in-Parallel analysis in place of Z3 to detect data races. In contrast, LLOV uses RDG (Reduced Dependence Graph) to determine parallelism in a region and infer data races based on the presence of data dependencies.

Other static analysis tools like Relay [90], Locksmith [78], and RacerX [31] use Eraser's [80] lockset algorithm and can detect races in `pthread`-based C/C++ programs. RacerD [13] is a static analysis tool for Java programs.

5.2 Dynamic Tools

Various dynamic race detection techniques have been proposed in the literature. The well known among them are based on Locksets [80], Happens-before [44] relations, and Offset-Span labels [63].

Archer [4] uses both static and dynamic analyses for race detection. It uses happens-before relations [44, 82], which enforces multiple runs of the program to find races. Archer reduces the analysis space of `pthread`-based tool TSan-LLVM by instrumenting only parallel sections of an OpenMP program. As Archer uses shadow memory to keep track of each memory access, memory requirement still remains the problem for memory bound programs.

In the analysis phase, Archer uses Polly to get the dependent loads and stores for a function and blacklist a section of code in the absence of dependencies. However, presence of dependence in a loop nest need not result in a data race.

```
1 #pragma omp parallel for
2 for (i=0;i<n;i++) {
3   for (j=1;j<m;j++) {
4     b[i][j]=b[i][j-1];
5   }
6 }
```

Listing 11. Loop nest with Loop Carried Dependence but without any data race.

The example in Listing 11 consists of a loop nest where the outer loop is parallel but the inner loop has a loop carried dependence. However, this is a valid parallelization strategy, since only the outer loop is marked parallel. With only loads and stores information, Archer will not be able to statically blacklist such code. It will have to rely on its dynamic analysis using TSan-LLVM. Moreover, the version of Archer (git master branch commit hash fc17353) used in our experiments completely disabled static analysis and does not use Polly at all. Archer only uses OMPT [30] callbacks to instrument the code with happens-before annotations for TSan-LLVM.

Since LLOV checks for parallelism at each level of the loop nest as stated in Section 4.2, hence it can statically detect the loop nest as data race free.

SWORD [5] is a dynamic tool based on operational semantic rules and uses OpenMP tools framework OMPT [30]. SWORD uses locksets to implement the semantic rules by taking advantage of the events tracked by OMPT. SWORD logs runtime traces for each thread consisting of all the OpenMP events and memory accesses using OMPT APIs. In the second offline phase, it analyzes the traces for concurrent threads using Offset-Span labels and detects unsynchronized memory accesses in two concurrent threads. If no synchronization is used on a common memory access, then a data race is flagged. SWORD cannot detect races in OpenMP SIMD, tasks and target offloading constructs.

ROMP [39] is a dynamic data race detection tool for OpenMP programs. ROMP maintains access history for each memory access. An access history consists of the access event, access type, and any set of locks associated with the access. ROMP constructs a task graph for the implicit and explicit OpenMP tasks and analyzes concurrent events. If an access event is concurrent and the memory is not protected by mutual exclusion mechanisms, then ROMP flags data race warnings. ROMP builds upon the offset-span-labels of OpenMP threads and constructs task graphs to detect races.

Helgrind [88] is a dynamic data race detection tool built in Valgrind framework [66] for C/C++ multithreaded programs. Helgrind maintains *happens-before* relations for each pair of memory accesses and forms a directed acyclic graph (DAG). If there is no path from one location to another in the happens-before graph, then data race is flagged.

Valgrind DRD [87] is another dynamic race detection tool in Valgrind. It can detect races in multithreaded C/C++ programs. It is based on happens-before relations similar to Helgrind.

ThreadSanitizer [82] is a dynamic data race detection tool based on Valgrind for multithreaded C, C++ programs. It employs a hybrid approach of happens-before annotations and maintains locksets for read and write operations on shared memory. It maintains a state machine as metadata called shadow memory. It reports a data race when two threads access the same memory location and their corresponding locksets are disjoint. Because of the shadow memory requirement for each memory access, ThreadSanitizer's memory requirement grows linearly with the amount of memory shared among threads. Binary instrumentation also increases the runtimes by around 5×-30× [83].

TSan-LLVM [83] is based on ThreadSanitizer [82]. TSan-LLVM uses LLVM to instrument the binaries in place of Valgrind. TSan-LLVM instrumented binaries incur less runtime overhead compared to ThreadSanitizer. However, it still has similar memory requirements and remains a bottleneck for larger programs.

Intel Inspector [40] is a commercial, dynamic data race detection tool for C, C++, and FORTRAN programs.

There are other dynamic analysis tools like Eraser [80], FastTrack [34], CoRD [42], and RaceTrack [92] that use the lockset algorithm to detect races in parallel programs. IFRit [29] is a dynamic race detection tool based on Interference Free Regions(IFR). Since they are not specific to OpenMP, we have not discussed them here.

OpenMP-aware tools: Majority of the tools are either POSIX thread based or are specific to race detection in inherent parallelism of various programming languages, such as Java, C#, X10, and Chappel. The tools ompVerify [8], Archer [4], PolyOMP [20], DRACO [91], SWORD [91], and ROMP [39] are the only ones that exploit the intricate details of structured parallelism in OpenMP.

OMPRacer [84] is another recent LLVM-based tool that was announced during the course of publication of our work. OMPRacer relies on alias analysis, happens-before relations and locksets to statically detect races in OpenMP programs. The published version of OMPRacer [84] has a comparison with an initial version of LLOV. The comparison is limited to only C/C++ benchmarks, with no mention of comparison using FORTRAN benchmarks. Comparing with their published

Table 2. Race Detection Tools with the Version Numbers Used for Comparison

Tools	Source	Version / Commit
Helgrind [88]	Valgrind	3.13.0
Valgrind DRD [87]	Valgrind	3.13.0
TSan-LLVM [82]	LLVM	6.0.1
Archer [4]	git master branch	fc17353
SWORD [5]	git master branch	7a08f3c
ROMP [39]	git master branch	6a0ad6d

numbers, the current version of LLOV outperforms OMPRacer in precision, recall, and accuracy on DataRaceBench v1.2 benchmark.

Polyhedral model-based static analysis tools: To the best of our knowledge, ompVerify [8], DRACO [91], PolyOMP [20], and LLOV are the only tools for race detection in OpenMP programs that are based on the polyhedral framework. To put it theoretically, these are the only tools that use the exact dependence analysis of polyhedral compilation, which is crucial for the exact analysis of a large class of useful loop programs [10].

LLOV is different from these tools as it works on the LLVM-IR and collects OpenMP pragmas that have been lowered to library calls. This makes LLOV language independent. Also, the analysis phase of LLOV could be used for other purposes, like generating task graphs from the LLVM-IR.

Race detection in OpenMP programs for GPUs: Multitude of works [11, 46–50, 67, 71, 94, 95] have investigated the problem of verification of Compute Unified Device Architecture (CUDA) programs. However, not much work has gone into verification of OpenMP device offloading constructs. This might be due to lack of complete support for OpenMP device offloading to CUDA devices by the mainstream compilers. The recent work from Barua et al. [6] performs verification of the OpenMP host-device data mapping with the *map* clause. Arm DDT [3] is a commercial debugging tool that supports debugging of OpenMP and CUDA programs.

Recent works by Liao et al. [51, 53] and Lin et al. [54] compare different race detection tools using the DataRaceBench [51] benchmark. In Section 6, we show comparison of our tool LLOV with the other race detection tools on DataRaceBench, as well as two other benchmarks in greater detail.

6 EXPERIMENTAL RESULTS

In this section, we describe our experimental setup, provide details on our experiments, and compare the results with other tools on a set of benchmarks.

6.1 Experimental Setup

We compare LLOV with the state-of-the-art data race detection tools as listed in Table 2. Some of the relevant tools were left out of our experimentation either because of their unavailability or due to their inability to handle OpenMP programs. We had issues setting up ROMP [39] and hence could not consider it for our comparison. ompVerify [8] is a prototype implementation in the Eclipse CDT/CODAN framework and can detect races in *omp parallel for* constructs only. Neither the binary nor source code for PolyOmp [20] and DRACO [91] are available in the open. We did not consider Intel Inspector due to its proprietary nature and licensing issues.[4]

[4]We could not procure an educational license for Intel ICC.

For evaluation we chose DataRaceBench v1.2 [52, 53], DataRaceBench FORTRAN [43] (a FORTRAN implementation of DataRaceBench v1.2), and OmpSCR v2.0 [26] benchmark suits. All these benchmarks use OpenMP for parallelization and have known data races. The benchmarks cover OpenMP v4.5 pragmas comprehensively and contain data races due to common mistakes in OpenMP programming as listed earlier (Section 3).

DataRaceBench v1.2 [51], a *seeded* OpenMP benchmark with known data races, consists of 116 microbenchmark kernels, of which 59 kernels have true data races, while the remaining 57 kernels do not have any data races. Of these 59 true race kernels, 44 kernels have exactly one data race. The remaining 15 have more than one data race due to read and write operations to a scalar variable inside a loop. Such cases have all the three types of dependencies, namely read after write (RAW), write after read (WAR), and write after write (WAW) dependencies and thus result in more than one race. We have considered the location of the write operation for these cases and hence considered one TP per kernel. DataRaceBench FORTRAN has 52 kernels with true data races and 40 kernels without any data races. OmpSCR v2.0 is a benchmark suite for high performance computing using OpenMP v3.0 APIs. The benchmark consists of C/C++ and FORTRAN kernels that demonstrate the usefulness and pitfalls of the parallel programming paradigm with both correct and incorrect parallelization strategies. The kernels range from parallelization of simple loops with dependencies to more complex parallel implementations of algorithms such as Mandelbrot set generator, Molecular Dynamics simulation, Pi (π) calculation, LU decomposition, Jacobi solver, fast Fourier transforms (FFT), and Quicksort.

Performance Metrics Notations. We define terminology used for performance metrics as follows:

- **True Positive (TP):** If the evaluation tool correctly detects a data race present in the kernel, then it is a True Positive test result. A higher number of true positives represents a better tool.
- **True Negative (TN):** If the benchmark does not contain a race and the tool declares it as race-free, then it is a true-negative case. A higher number of true negatives represents a better tool.
- **False Positives (FP):** If the benchmark does not contain any race but the tool reports a race condition, then it is a false-positive case. A lower number of false positives are desirable.
- **False Negatives (FN):** False Negative test result is obtained when the tool fails to detect a known race in the benchmark. These are the cases that are missed by the tool. A lower number of false negatives are desirable.

We consider the following statistical measures as performance metrics in our experiments.

- **Precision:** Precision is the measure of closeness of the outcomes of prediction. Thus, a higher value of precision represents that the tool will more often than not identify a race condition when it exists.
  $Precision = \frac{TP}{TP + FP}$ $Recall = \frac{TP}{TP + FN}$.
- **Recall:** Recall gives the total number of cases detected out of the maximum data races present. A higher recall value means that there are less chances that a data race is missed by the tool. It is also called true-positive rate (TPR).
- **Accuracy:** Accuracy gives the chances of correct reports out of all the reports, as the name suggests. A higher value of accuracy is always desired and gives overall measure of the efficacy of the tool.
  $Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$ $F1\ Score = 2 * \frac{Precision * Recall}{Precision + Recall}$.

Table 3. Maximum Number of Races Reported by Different Tools in DataRaceBench 1.2

Tools	Race: Yes		Race: No		Coverage/116
	TP	FN	TN	FP	
Helgrind	56	3	2	55	116
Valgrind DRD	56	3	26	31	116
TSan-LLVM	**57**	2	2	55	116
Archer	56	3	2	55	116
SWORD	47	4	24	4	79
LLOV	48	2	**36**	5	91

- **F1 Score:** The harmonic mean of precision and recall is called the F1 score. An F1 score of 1 can be achieved in the best case when both precision and recall are perfect. The worst-case F1 score is 0 when either precision or recall is 0.
- **Diagnostic odds ratio (DOR):** It is the ratio of the positive likelihood ratio (LR+) to the negative likelihood ratio (LR−).
  $DOR = \frac{LR+}{LR-}$ where,
  Positive Likelihood Ratio $(LR+) = \frac{TPR}{FPR}$, False Positive Rate $(FPR) = \frac{FP}{FP + TN}$,
  Negative Likelihood Ratio $(LR-) = \frac{FNR}{TNR}$, False Negative Rate $(FNR) = \frac{FN}{FN + TP}$ and
  True Positive Rate $(TPR) = \frac{TP}{TP + FN}$, True Negative Rare $(TNR) = \frac{TN}{TN + FP}$.
  DOR is the measure of the ratio of the odds of race detection being positive given that the test case has a data race, to the odds of race detection being positive given the test case does not have a race.

System configuration. We performed all our experiments on a system with two Intel Xeon E5-2697 v4 @ 2.30 GHz processors, each having 18 cores and 2 threads per core, totalling 72 threads and 128GB of RAM. The system runs 64 bit Ubuntu 18.04.2 LTS server with Linux kernel version 4.15.0-48-generic. LLOV is currently based on the LLVM/Polly version release 7.0.1 and can be upgraded to the latest LLVM/Polly versions with minimal changes.

Similar to Liao et al. [51], our experiments use two parameters: (i) the number of OpenMP threads and (ii) the input size for variable length arrays. The number of threads that we considered for the experiments are {3, 36, 45, 72, 90, 180, 256}. For the 16 variable length kernels, we considered 6 different array sizes as follows: {32, 64, 128, 256, 512, 1024}. With each particular set of parameters, we ran each of the 116 kernels 5 times. Both the number of threads and array sizes can be found in prior studies [51, 53] and we have used the same for uniformity. Since the dynamic tools depend on the execution order of the threads, multiple runs are required. The 16 kernels with variable length arrays were run 3,360 (16 kernels × 7 thread sizes × 6 array sizes × 5 runs) times in total. The remaining 100 kernels were run 3,500 (100 kernels × 7 thread sizes × 5 runs) times in total. For all experiments, we used a timeout of 600 s for compilation as well as execution separately.

6.2 Experimental Results

6.2.1 DataRaceBench 1.2. Table 3 provides comparison in terms of the number of races detected in DataRaceBench v1.2. Column 1 indicates the name of the tool. The column with titles "Race:Yes" and "Race:No" indicates if the benchmark had a race or not. The subcolumns "TP" and "FN" denote whether the tool was able to find the race or not when the benchmark had a race. Similarly, the subcolumns "FP" and "TN" denote if the tool erroneously reported a race or reported the absence

Table 4. Precision, Recall, and Accuracy of the Tools on DataRaceBench 1.2

Tools	Precision	Recall	Accuracy	F1 Score	Diagnostic odds ratio
Helgrind	0.50	0.95	0.50	0.66	0.68
Valgrind DRD	0.64	0.95	0.71	0.77	15.66
TSan-LLVM	0.51	**0.97**	0.51	0.67	1.04
Archer	0.50	0.95	0.50	0.66	0.68
SWORD	**0.92**	0.92	0.90	0.92	70.50
LLOV	0.91	0.96	**0.92**	**0.93**	**172.80**

Table 5. Maximum Number of Races Reported by Different Tools in Common 61 Kernels of DataRaceBench 1.2

Tools	Race: Yes		Race: No		Coverage/61
	TP	FN	TN	FP	
Helgrind	42	1	2	16	61
Valgrind DRD	42	1	12	6	61
TSan-LLVM	42	1	2	16	61
Archer	42	1	2	16	61
SWORD	42	1	17	1	61
LLOV	42	1	16	2	61

of a race when the benchmark did not have a race. Dynamic analysis tools are run multiple times, and, if the tool reports a race in any of the runs, then, it is considered that the tool will classify the benchmark as having a race.

LLOV could analyze 91 of 116 (78.45%) kernels from the benchmark and could detect 48 TP and 36 TN. As the analysis of our tool is conservative, it also produces 5 FP. Moreover, it had 2 FN because of inter SCoP races that is a limitation of the current version of LLOV.

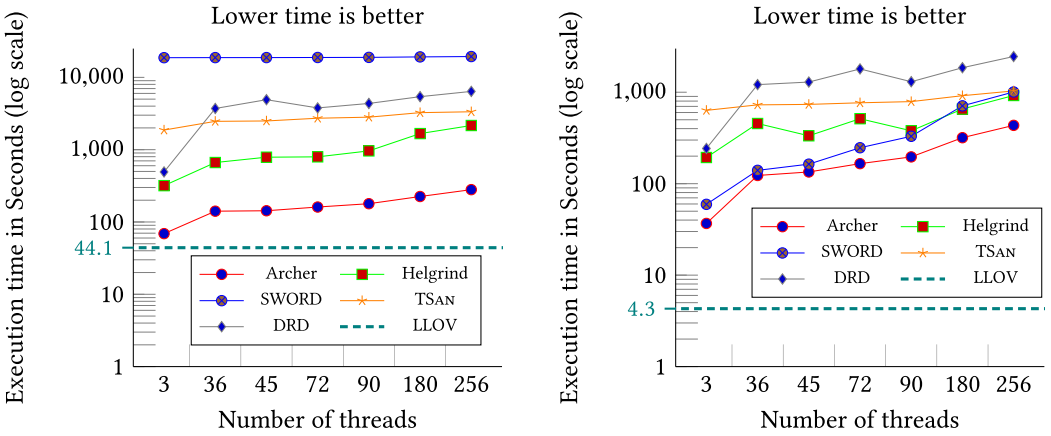
Due to the sound static analysis that LLOV implements, it could also prove that 36 of the kernels are *data race free*. LLOV is unique in this regard, other tools *are not able to make* such a claim. LLOV will report one of the following three cases: *data race detected* when LLOV detects a race, *data race free* when LLOV can statically prove that the parallel segment of the input program does not have dependencies due to shared memory accesses, and finally, *region not analyzed* when LLOV cannot analyze the input program. In addition, LLOV also reports if an OpenMP pragma is not supported (refer to Table 1). Due to these reasons, as of now LLOV does not provide complete coverage on DataRaceBench v1.2. SWORD provides even lesser coverage on DataRaceBench v1.2 due to its compilation related issues.

Table 4 shows performance of the tool on various performance metrics defined earlier in the section. From Table 4 it appears that LLOV's performance is the best followed by SWORD in second in terms of precision, accuracy, F1 score, and DOR. Since both SWORD and LLOV do not have complete coverage, a more appropriate strategy would be to compare all the tools on only those benchmarks that they are able to handle/cover.

Table 5 shows how tools classify benchmarks with respect to data race on 61 benchmarks that all the tools are able to handle/cover. Table 6 provides a comparison of the tools on 61 benchmarks on various performance metrics. It is indeed the case that on the benchmarks that SWORD is able to handle, it achieves the highest precision, accuracy, recall, F1 score, and diagnostic odds ratio. LLOV is a close second with respect to precision, accuracy, and F1 score. One must keep in mind that both SWORD and LLOV may gain advantage in terms of these metrics because of lesser coverage. A

Table 6. Precision, Recall, and Accuracy of the Tools on Common 61 Kernels of DataRaceBench 1.2

Tools	Precision	Recall	Accuracy	F1 Score	Diagnostic odds ratio
Helgrind	0.72	**0.98**	0.72	0.83	5.25
Valgrind DRD	0.88	**0.98**	0.89	0.92	84.00
TSan-LLVM	0.72	**0.98**	0.72	0.83	5.25
Archer	0.72	**0.98**	0.72	0.83	5.25
SWORD	**0.98**	**0.98**	**0.97**	**0.98**	**714.00**
LLOV	0.95	**0.98**	0.95	0.97	336.00



(a) Time taken on all 116 kernels

(b) Time taken on common 61 kernels

Fig. 3. DataRaceBench v1.2 total time taken on logarithmic scale.

crucial point to note is that while SWORD crashes on many benchmarks, LLOV provides graceful reporting and exit on benchmarks it is not able to cover, providing a better user experience.

Though LLOV does not come out on top on various metrics such as coverage, precision, and so on, it completely outshines other tools in terms of runtime. Figure 3(a) shows the performance of the tool with respect to the runtime. Since dynamic tools run benchmarks multiple times we report the average time taken for each benchmark, and the total time is the sum of these averages. In Figure 3(a), y-axis represents total time taken in seconds by a tool on a logarithmic scale. Time taken by LLOV to analyze all 116 kernels is a mere 44.1 s. However, other tools take orders of magnitude more time as compared to LLOV. The reason for SWORD performing the worst when all 116 benchmarks are considered is because the compilation process itself times out for several benchmarks. Timeout value of 600 s is used for each kernel for both compilation and execution separately. For LLOV the only time required is the time to compile as it does its analysis at compile time. In addition, as LLOV is able to detect the cases it can not analyze, the exits for such programs are graceful. The power of static analysis in LLOV is particularly evident in Figure 3 as the time taken remains constant irrespective of the number of threads.

The Polyhedral framework is known for large compile-time overheads, because it relies on computationally expensive algorithms for dependence analysis, scheduling and code-generation. These algorithms could be exponential, or polynomials of high-degree in complexity in the number of dimensions in the loop nest [1, 7, 32, 86, 89]. However, very few programs in the real world have very large loop-depths.

Table 7. Maximum Number of Races Reported by Different Tools in DataRaceBench FORTRAN

Tools	Race: Yes		Race: No		Coverage/92
	TP	FN	TN	FP	
Helgrind	**46**	6	4	36	92
Valgrind DRD	45	7	**21**	19	92
LLOV	36	7	19	5	67

In DataRaceBench v1.2, there are six tiled and parallel versions of PolyBench/C 3.2 kernels. The tiled version of matrix multiplication kernel DRB042-3mm-tile-no has 408 OpenMP parallel loops and contributes to around 69.21% of total time taken by LLOV for all the 116 kernels. Although such kernels are not very common in real-world scenarios, as they are generated by polyhedral tools such as PLuTo [16], verification of code generated by such automatic tools remains a challenge.

Figure 3(b) shows the runtime performance of the tools on 61 benchmarks that all the tools are able to cover. On this subset of benchmarks, LLOV outperforms all the other tools by orders of magnitude. Archer is second in performance, and SWORD comes third.

6.2.2 DataRaceBench 1.2 FORTRAN. Since LLOV is based on LLVM-IR, it is language independent. To demonstrate this, we reimplemented DataRaceBench 1.2 in FORTRAN 95 [43] rewriting 92 of the 116 DataRaceBench v1.2 kernels in FORTRAN. The other kernels, such as {41, 42, 43, 44, 55, 56}, are Polybench kernels that were tiled and/or parallelized by the POCC [77] polyhedral tool, and are not amenable to easy re-writing in FORTRAN.

The kernel in Listing 2 (Section 3) is from DataRaceBench 1.2 FORTRAN that has a data race because of the write to `shared` variable `tmp` (Line 3) and the read from `tmp` (Line 4). The race in this example can be avoided by explicitly stating that `tmp` is a private variable for each thread using the `private` clause. LLOV could detect such races because of missing data sharing clauses, such as `private`, `reduction`, `firstprivate`, and `lastprivate`.

To verify these FORTRAN kernels, we used LLVM FORTRAN frontend Flang [61], which is under active development and has officially been accepted in the year 2019 as a sub-project under the LLVM Compiler Infrastructure umbrella project. We used Flang version 7.1.0 (git commit hash cb42a171) to generate LLVM-IR from FORTRAN source code and ran LLOV on the generated IR.

Initial experiments show that our analysis is able to detect data races in OpenMP kernels of DataRaceBench FORTRAN. To the best of our knowledge, LLOV is the *only static tool* to be able to detect races in OpenMP programs written in FORTRAN. Table 7 shows that LLOV could detect 36 TP and also confirm that 19 kernels are data race free (TN). LLOV also produced 5 FP along with 7 FN.

Kernels {72, 78, 79, 94, 112} are not analyzed by LLOV, because the current version of Flang does not support the corresponding OpenMP directives. This explains the difference in the numbers from DataRaceBench v1.2 (Table 3) and DataRaceBench FORTRAN (Table 7). Flang produced segmentation faults for the kernels {84, 85} having OpenMP `threadprivate` variables. As Flang is in active development, we believe that more OpenMP directives will be supported in the upcoming releases. Another challenge we faced is detecting polyhedral SCoPs in Flang generated IR using Polly [62]. Even for functionally equivalent source codes, the IR generated by Clang and Flang differs considerably. So, the native implementations of analyses and optimization passes in LLVM need to be modified to have uniform results on LLVM-IR generated by Clang and Flang.

Table 8. Number of Races Detected in OmpSCR v2.0 (CT is Compilation Timeout, NA for Not Analyzed)

Kernel	LLOV	Helgrind	DRD	TSan-LLVM	Archer	SWORD
Manually verified kernels with data races						
c_loopA.badSolution	1	1	1	1	1	1
c_loopA.solution2	NA	1	1	1	1	0
c_loopA.solution3	1	1	1	1	1	0
c_loopB.badSolution1	1	1	1	1	1	1
c_loopB.badSolution2	1	1	1	1	1	1
c_loopB.pipelineSolution	NA	1	1	1	1	0
c_lu	NA	1	1	1	1	0
c_jacobi03	1	1	1	0	0	CT
Manually verified race free kernels						
c_loopA.solution1	0	2	1	2	1	0
c_md	1	2	2	2	1	CT
c_mandel	NA	1	0	1	1	0
c_pi	0	1	0	1	1	0
c_jacobi01	2	2	1	0	0	CT
c_jacobi02	1	1	1	0	0	CT
Unverified kernels						
c_fft	0	1	1	1	1	CT
c_fft6	2	1	0	1	1	CT
c_qsort	0	1	1	1	1	CT
c_GraphSearch	0	0	0	0	0	0
cpp_qsomp1	0	0	0	0	0	0
cpp_qsomp2	0	0	0	0	0	0
cpp_qsomp3	0	0	0	0	0	0
cpp_qsomp4	0	0	0	0	0	0
cpp_qsomp5	1	0	0	0	0	0
cpp_qsomp6	0	0	0	0	0	0
cpp_qsomp7	0	0	0	0	0	0

6.2.3 OmpSCR v2.0. We also evaluated all the tools listed in Table 2 on OmpSCR v2.0 [26, 27] kernels. For the dynamic tools, we used default program parameters provided with argument *-test*. We have done minor modifications in OmpSCR v2.0 [26] to compile it on the latest operating systems with updated system calls and created scripts to run and test all the data race checkers. The updated version of OmpSCR can be found in Reference [28].

We manually verified the OmpSCR v2.0 benchmark suite. Table 8 divides the benchmarks into three categories: (1) Manually verified kernels with data races, (2) Manually verified race-free kernels, and (3) Unverified kernels. Of 25 kernels, 11 remained unverified due to various complexities such as recursive calls using OpenMP pragmas. Every cell in Table 8 denotes how many different regions are reported as containing a race by a tool. Every verified kernel having a race contains only one parallel region containing races. All the races reported by a tool belonging to a single parallel region is counted as only one. The reason for combining all the races in a region is because otherwise the number of races reported for dynamic tools becomes quite high (e.g., several races reported for a single array). Some of the programs contain multiple parallel regions. If a tool reports races in two distinct regions, then we count the tool as reporting two races. Since

Table 9. Comparison of Different Tools on OmpSCR v2.0

Tools	Race: Yes		Race: No		Coverage/14
	TP	FN	TN	FP	
Helgrind	8	0	0	9	14
Valgrind DRD	8	0	2	5	14
TSan-LLVM	7	1	2	6	14
Archer	7	1	2	4	14
SWORD	3	4	3	0	10
LLOV	4	1	2	5	10

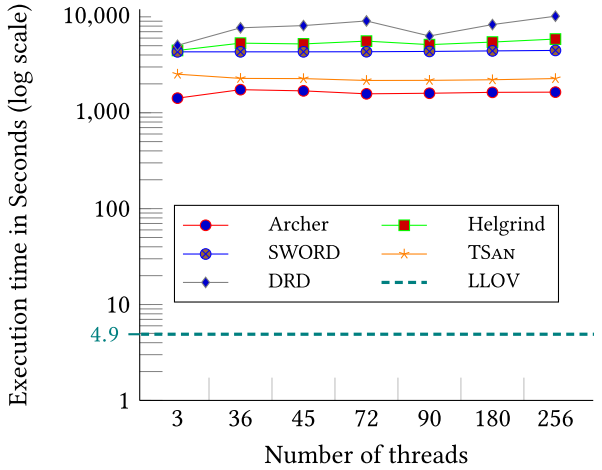


Fig. 4. OmpSCR v2.0 total execution time by different tools on logarithmic scale.

every verified kernel with race has only one parallel region having a race, races reported in other race-free regions are counted toward *false positives*.

As shown in Table 8, LLOV is able to detect true data races in c_loopA and c_loopB kernels. LLOV produced a false negative for Jacobi03 kernel due to the presence of a dependence across two SCoPs, which is a limitation of the current version of LLOV. Our tool produced false positives for all three Jacobi kernels due to conservative Mod/Ref analysis in LLVM. All the three Jacobi kernels in OmpSCR use a one-dimensional array that is passed to the kernel as a pointer. Also, the programs are written with the arrays accessed in the column-major order, which means that their array subscripts are incorrectly computed in C/C++. Because of these reasons, the three Jacobi kernels are not modelled by Polly even though Jacobi is a standard polyhedral kernel. Table 9 summarizes the races detected by different tools in OmpSCR v2.0.

All the false positives flagged by LLOV are because of `shared` double pointer variables. In addition, SWORD ends up with compiler timeout (CT) for kernels such as Molecular Dynamics, Quicksort (c_qsort), FFT, and Jacobi. The time taken to detect races in all the kernels by the tools is shown in Figure 4. It can be seen that LLOV completes its analysis for the entire benchmark in just *4.9 s*, while the other state-of-the-art tools take orders of magnitude longer.

7 CONCLUSIONS AND FUTURE WORK

In this article, we present LLOV, a language agnostic, OpenMP-aware, static analysis-based data race detection tool that is developed on top of the LLVM compiler framework. As LLOV operates

at LLVM-IR level, it can support a multitude of programming languages supported by the LLVM infrastructure. We successfully demonstrate the language agnostic nature of LLOV by performing data race checks on a standard set of benchmarks written in C, C++, and FORTRAN.

Our experiments show that LLOV performs reasonably well in terms of precision and accuracy while being *most performant* with respect to other tools by a large margin. Though at present, LLOV supports only some of the pragmas offered by OpenMP, it gracefully exits on input programs that contain pragmas that it is unable to handle. We would like to further enrich LLOV by adding support for various OpenMP pragmas that are not supported at present. Many such pragmas offer an engineering challenge as the structural information is not available at LLVM-IR level, and such information has to be reconstructed from the IR.

Our tool is primarily based on the polyhedral compilation framework, Polly. The use of approximate dependence analysis [57] readily available in LLVM may further increase the capability and scalability of LLOV. We also plan to extend the support for dynamic control flow and irregular accesses using the extended polyhedral framework [9, 10, 22]. We plan to use May-Happen-in-Parallel (MHP) analysis to provide coverage for OpenMP tasks, synchronizations, and sections constructs and overcome the current limitation of FN cases for dependencies across SCoPs.

The tool and other relevant material are available at http://compilers.cse.iith.ac.in/projects/llov.

ACKNOWLEDGMENTS

We thank Tobias Grosser, Johannes Doerfert, and Michael Kruse for their help with the initial version of Polly as an analysis pass, which we extended for this work. We also thank Govindarajan Ramaswamy and V. Krishna Nandivada for their feedback on this work. We thank the anonymous reviewers of ACM TACO for their insightful comments that helped in improving the article.

REFERENCES

[1] A. Acharya, U. Bondhugula, and A. Cohen. 2018. Polyhedral auto-transformation with no integer linear programming. In *Proceedings of the 39th ACM SIGPLAN Conference on PLDI*. ACM, New York, NY, 529–542.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (2nd Ed.). Addison-Wesley Longman, Boston, MA.

[3] ARM. 2020. ARM DDT. Retrieved July 21, 2020 from https://www.arm.com/products/development-tools/server-and-hpc/forge/ddt.

[4] S. Atzeni, G. Gopalakrishnan, Z. Rakamaric, D. H. Ahn, I. Laguna, M. Schulz, G. L. Lee, J. Protze, and M. S. Müller. 2016. ARCHER: Effectively spotting data races in large OpenMP applications. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'16)*. IEEE, USA, 53–62.

[5] S. Atzeni, G. Gopalakrishnan, Z. Rakamaric, Ignacio Laguna, Gregory L. Lee, and D. H. Ahn. 2018. Sword: A bounded memory-overhead detector of OpenMP data races in production runs. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'18)*. IEEE, 845–854.

[6] P. Barua, J. Shirako, W. Tsang, J. Paudel, W. Chen, and V. Sarkar. 2019. OMPSan: Static verification of OpenMP's data mapping constructs. In *OpenMP: Conquering the Full Hardware Spectrum*. Springer, 3–18.

[7] Cédric Bastoul. 2004. Code generation in the polyhedral model is easier than you think. In *Proceedings of the IEEE International Conference on Parallel Architecture and Compilation Techniques (PACT'13)*. France, 7–16.

[8] V. Basupalli, T. Yuki, S. Rajopadhye, A. Morvan, S. Derrien, P. Quinton, and D. Wonnacott. 2011. ompVerify: Polyhedral analysis for the OpenMP programmer. In *Proceedings of the International Workshop on OpenMP*. Springer, Berlin, 37–53.

[9] Marouane Belaoucha, Denis Barthou, Adrien Eliche, et al. 2010. FADAlib: An open source C++ library for fuzzy array dataflow analysis. *Proc. Comput. Sci.* 1, 1 (2010), 2075–2084.

[10] M. W. Benabderrahmane, L. N. Pouchet, A. Cohen, and C. Bastoul. 2010. The polyhedral model is more widely applicable than you think. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction (CC'10/ETAPS'10)*. Springer-Verlag, Berlin, 283–303.

[11] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson. 2012. GPUVerify: A verifier for GPU kernels. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'12)*. ACM, NY, 113–132. DOI: https://doi.org/10.1145/2384616.2384625

[12] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2017. Julia: A fresh approach to numerical computing. *SIAM Rev.* 59, 1 (2017), 65–98. DOI: https://doi.org/10.1137/141000671

[13] S. Blackshear, N. Gorogiannis, P. W. O'Hearn, and I. Sergey. 2018. RacerD: Compositional static race detection. *Proc. ACM Program. Lang.* 2, Article 144 (Oct. 2018), 28 pages. DOI: https://doi.org/10.1145/3276514

[14] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. 1996. Cilk: An efficient multithreaded runtime system. *J. Parallel Distrib. Comput.* 37, 1 (1996), 55–69.

[15] Uday Bondhugula. 2017. PLUTO—An Automatic Parallelizer and Locality Optimizer for Affine Loop Nests. Retrieved May 15, 2020 from http://pluto-compiler.sourceforge.net/.

[16] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. ACM, New York, NY, 101–113. DOI: https://doi.org/10.1145/1375581.1375595

[17] Utpal Bora, Johannes Doerfert, Tobias Grosser, and Ramakrishna Upadrasta. 2016. GSoC 2016: PolyhedralInfo—Polly as an Analysis Pass in LLVM. Retrieved May 8, 2019 from https://llvmdevelopersmeetingbay2016.sched.com/event/8Z2Z/lightning-talks.

[18] B. L. Chamberlain, D. Callahan, and H. P. Zima. 2007. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.* 21, 3 (August 2007), 291–312. DOI: https://doi.org/10.1177/1094342007078442

[19] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. 2005. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*. ACM, New York, NY, 519–538.

[20] P. Chatarasi, J. Shirako, M. Kong, and V. Sarkar. 2016. An extended polyhedral model for SPMD programs and its use in static data race detection. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*. Springer, 106–120.

[21] P. Chatarasi, J. Shirako, and V. Sarkar. 2016. Static data race detection for SPMD programs via an extended polyhedral representation. In *Proceedings of the 6th International Workshop on Polyhedral Compilation Techniques (IMPACT'16)*.

[22] J.-F. Collard, D. Barthou, and P. Feautrier. 1995. Fuzzy array dataflow analysis. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'95)*. ACM, New York, NY, 92–101.

[23] Cray. 2019. Chapel Language Specification (Version 0.9). Retrieved from http://chapel.cray.com/papers.html.

[24] Leonardo Dagum and Rameshm Enon. 1998. OpenMP: An industry standard API for shared-memory programming. *IEEE Comput. Sci. Eng.* 5, 1 (1998), 46–55.

[25] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, Berlin, 337–340.

[26] A. J. Dorta, C. Rodriguez, and F. de Sande. 2004. OpenMP Source Code Repository. Retrieved May 19, 2019 from https://sourceforge.net/projects/ompscr/files/OmpSCR/.

[27] A. J. Dorta, C. Rodriguez, and F. de Sande. 2005. The OpenMP source code repository. In *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*. IEEE, Los Alamitos, CA, 244–250. DOI: https://doi.org/10.1109/EMPDP.2005.41

[28] A. J. Dorta, C. Rodriguez, F. de Sande, and U. Bora. 2019. OpenMP Source Code Repository (Updates to Support Latest Compilers and Scripts). Retrieved May 19, 2019 from https://github.com/utpalbora/OmpSCR_v2.0.git.

[29] L. Effinger-Dean, B. Lucia, L. Ceze, D. Grossman, and H.-J. Boehm. 2012. IFRit: Interference-free regions for dynamic data-race detection. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'12)*. ACM, New York, NY, 467–484.

[30] A. E. Eichenberger, J. Mellor-Crummey, M. Schulz, M. Wong, N. Copty, R. Dietrich, X. Liu, E. Loh, and D. Lorenz. 2013. OMPT: An OpenMP tools application programming interface for performance analysis. In *OpenMP in the Era of Low Power Devices and Accelerators*. Springer, Berlin, 171–185.

[31] D. Engler and K. Ashcraft. 2003. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*. ACM, New York, NY, 237–252.

[32] Paul Feautrier. 1988. Parametric integer programming. *RAIRO-Operat. Res.* 22, 3 (1988), 243–268.

[33] Paul Feautrier. 1991. Dataflow analysis of array and scalar references. *Int. J. Parallel Program.* 20, 1 (1991), 23–53. DOI: https://doi.org/10.1007/BF01407931

[34] C. Flanagan and S. N. Freund. 2009. FastTrack: Efficient and precise dynamic race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*. 121–133.

[35] Project GNU. 2019. GCC, the GNU Compiler Collection. Retrieved from August 8, 2019 from https://gcc.gnu.org/.

[36] Tobias Grosser, Armin Größlinger, and Christian Lengauer. 2012. Polly - Performing polyhedral optimizations on a low-level intermediate representation. *Parallel Process. Lett.* 22, 4, Article 1 (2012), 27 pages. DOI: https://doi.org/10.1142/S0129626412500107

[37] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. 2011. Polly-polyhedral optimization in LLVM. In *Proceedings of the 1st International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, Vol. 2011. IEEE, IEEE Computer Society, Los Alamitos, CA, 1.

[38] Khronos OpenCL Working Group. 2019. The OpenCL Specification, v2.2-11 (July 2019). Retrieved August 8, 2019 from https://www.khronos.org/registry/OpenCL/.

[39] Y. Gu and J. Mellor-Crummey. 2018. Dynamic data race detection for OpenMP programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'18)*. IEEE Press, Los Alamitos, CA, Article 61, 12 pages.

[40] Intel. 2019. Intel Inspector. Retrieved May 8, 2019 from https://software.intel.com/en-us/inspector.

[41] Intel. 2019. Intel Threading Building Blocks. Retrieved August 8, 2019 from https://software.intel.com/en-us/tbb.

[42] B. Kasikci, C. Zamfir, and G. Candea. 2012. CoRD: A collaborative framework for distributed data race detection. In *Proceedings of the 8th USENIX Conference on Hot Topics in System Dependability (HotDep'12)*. USENIX, 4–4.

[43] Pankaj Kukreja, Himanshu Shukla, and Utpal Bora. 2019. DataRaceBench FORTRAN. Retrieved October 19, 2019 from https://github.com/IITH-Compilers/drb_fortran.

[44] L. Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (July 1978), 558–565. DOI: https://doi.org/10.1145/359545.359563

[45] N. G. Leveson and C. S. Turner. 1993. An investigation of the Therac-25 accidents. *Computer* 26, 7 (July 1993), 18–41. DOI: https://doi.org/10.1109/MC.1993.274940

[46] G. Li and G. Gopalakrishnan. 2010. Scalable SMT-based verification of GPU kernel functions. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'10)*. ACM, New York, NY, 187–196.

[47] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan. 2012. GKLEE: Concolic verification and test generation for GPUs. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*. Association for Computing Machinery, New York, NY, 215–224. DOI: https://doi.org/10.1145/2145816.2145844

[48] P. Li, X Hu, D. Chen, J. Brock, H. Luo, E. Z. Zhang, and C. Ding. 2017. LD: Low-overhead GPU race detection without access monitoring. *ACM Trans. Archit. Code Optim.* 14, 1, Article 9 (March 2017), 25 pages.

[49] P. Li, G. Li, and G. Gopalakrishnan. 2012. Parametric flows: Automated behavior equivalencing for symbolic analysis of races in CUDA programs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*. 1–10.

[50] P. Li, G. Li, and G. Gopalakrishnan. 2014. Practical symbolic race checking of GPU programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*. 179–190.

[51] C. Liao, Pei-Hung Lin, J. Asplund, M. Schordan, and I. Karlin. 2017. 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)*. ACM, New York, NY, Article 11, 14 pages.

[52] Chunhua Liao, Pei-Hung Lin, Joshua Asplund, Markus Schordan, and Ian Karlin. 2018. DataRaceBench v1.2.0. Retrieved May 19, 2019 from https://github.com/LLNL/dataracebench.

[53] C. Liao, P.-H. Lin, M. Schordan, and I. Karlin. 2018. A semantics-driven approach to improving DataRaceBench's OpenMP standard coverage. In *Evolving OpenMP for Evolving Architectures*. Springer, 189–202.

[54] P. Lin, C. Liao, M. Schordan, and I. Karlin. 2019. Exploring regression of data race detection tools using DataRaceBench. In *Proceedings of the IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness'19)*. 11–18.

[55] Jacques-Louis Lions et al. 1996. Flight 501 failure. (1996).

[56] LLVM. 2019. LLVM Language Reference Manual. Retrieved August 8, 2019 from https://llvm.org/docs/LangRef.html.

[57] LLVM. 2019. Loop Access Info, Class Reference. Retrieved May 8, 2019 from https://llvm.org/doxygen/classllvm_1_1LoopAccessInfo.html.

[58] LLVM. 2019. The LLVM Compiler Infrastructure. Retrieved May 8, 2019 from http://llvm.org/.

[59] LLVM. 2020. LLVM Alias Analysis Infrastructure. Retrieved May 6, 2020 from https://llvm.org/docs/AliasAnalysis.html.

[60] LLVM/Clang. 2019. Clang: A C language family frontend for LLVM. Retrieved August 8, 2019 from https://clang.llvm.org.

[61] LLVM/Flang. 2019. Flang: A Fortran Compiler Targeting LLVM. Retrieved May 8, 2019 from https://github.com/flang-compiler/flang/wiki.

[62] LLVM/Polly. 2019. Polly: LLVM Framework for High-Level Loop and Data-Locality Optimizations. Retrieved May 8, 2019 from https://polly.llvm.org/.

[63] J. Mellor-Crummey. 1991. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (SC'91)*. ACM, New York, NY, 24–33. DOI: https://doi.org/10.1145/125826.125861

[64] MPI. 2019. MPI-3.1 (May 2019). Retrieved August 8, 2019 from https://www.mpi-forum.org/docs/.
[65] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. 2007. Automatically classifying benign and harmful data races using replay analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on PLDI*. ACM, New York, NY, 22–31. DOI: https://doi.org/10.1145/1250734.1250738
[66] N. Nethercote and J. Seward. 2003. Valgrind: A program supervision framework. Retrieved from http://valgrind.org/.
[67] NVIDIA. 2020. CUDA-MEMCHECK. Retrieved July 21, 2020 from https://docs.nvidia.com/cuda/cuda-memcheck/index.html.
[68] OpenACC. 2019. The OpenACC Application Programming Interface, v2.7 (November 2018). Retrieved August 8, 2019 from https://www.openacc.org/specification.
[69] OpenMP Architecture Review Board. 1997. OpenMP Application Programming Interface. Retrieved October 19, 2019 from https://www.openmp.org.
[70] OpenMP Architecture Review Board. 2015. OpenMP Application Programming Interface Version 4.5. Retrieved May 19, 2019 from https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.
[71] Y. Peng, V. Grover, and J. Devietti. 2018. CURD: A dynamic CUDA race detector. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)*. ACM, New York, NY, 390–403.
[72] PGI. 2018. PGI Accelerator Compilers with OpenACC Directives. Retrieved August 8, 2019 from https://www.pgroup.com/resources/accel.htm.
[73] Chuck Pheatt. 2008. Intel Threading building blocks. *J. Comput. Sci. Coll.* 23, 4 (April 2008), 298.
[74] Nathaniel Popper. 2012. Knight capital says trading glitch cost it $440 million. *New York Times* 2 (2012).
[75] Working Group for POSIX. 2018. IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7. *IEEE Std 1003.1-2017 (Rev. of IEEE Std 1003.1-2008)* (January 2018), 1–3951. DOI: https://doi.org/10.1109/IEEESTD.2018.8277153
[76] Louis-Noël Pouchet. 2012. PolyOpt/C: A Polyhedral Optimizer for the ROSE Compiler. Retrieved May 15, 2020 from http://web.cs.ucla.edu/ pouchet/software/polyopt/.
[77] Louis-Noël Pouchet. 2013. PoCC, the POlyhedral Compiler Collection Package: A Full Source-to-Source Polyhedral Compiler. Retrieved May 15, 2020 from https://sourceforge.net/projects/pocc/.
[78] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. 2011. LOCKSMITH: Practical static race detection for C. *ACM Trans. Program. Lang. Syst.* 33, 1 (2011), 3.
[79] Dan Quinlan and Chunhua Liao. 2011. The ROSE source-to-source compiler infrastructure. In *Proceedings of the Cetus Users and Compiler Infrastructure Workshop, in Conjunction with PACT*, Vol. 2011. Citeseer.
[80] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.* 15, 4 (1997), 391–411.
[81] Markus Schordan and Dan Quinlan. 2003. A source-to-source architecture for user-defined optimizations. In *Modular Programming Languages*. Springer, Berlin, 214–223.
[82] K. Serebryany and T. Iskhodzhanov. 2009. ThreadSanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA'09)*. ACM, New York, NY, 62–71. DOI: https://doi.org/10.1145/1791194.1791203
[83] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov. 2012. Dynamic race detection with LLVM compiler. In *Proceedings of the 2nd International Conference on Runtime Verification (RV'11)*. Springer-Verlag, 110–114.
[84] Bradley Swain, Yanze Li, Peiming Liu, Ignacio Laguna, Giorgis Georgakoudis, and Jeff Huang. 2020. OMPRacer: A scalable and precise static race detector for OpenMP programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'20)*.
[85] TOP500.Org. 2019. Top 500 Supercomputer Sites. Retrieved May 8, 2019 from http://www.top500.org/.
[86] R. Upadrasta and A. Cohen. 2013. Sub-polyhedral scheduling using (unit-)two-variable-per-inequality polyhedra. In *Proceedings of the 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.
[87] Valgrind-project. 2007. DRD: A Thread Error Detector. Retrieved May 8, 2019 from http://valgrind.org/docs/manual/drd-manual.html.
[88] Valgrind-project. 2007. Helgrind: A Thread Error Detector. Retrieved May 8, 2019 from http://valgrind.org/docs/manual/hg-manual.html.
[89] Sven Verdoolaege. 2010. Isl: An integer set library for the polyhedral model. In *Proceedings of the 3rd International Congress Conference on Mathematical Software (ICMS'10)*. Springer-Verlag, Berlin, Article 1250010, 4 pages.
[90] J. W. Voung, R. Jhala, and S. Lerner. 2007. RELAY: Static race detection on millions of lines of code. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, New York, NY, 205–214.
[91] F. Ye, M. Schordan, C. Liao, P. Lin, I. Karlin, and V. Sarkar. 2018. Using polyhedral analysis to verify openmp applications are data race free. In *Proceedings of the 2018 IEEE/ACM 2nd International Workshop on Software Correctness*

for HPC Applications (Correctness'18). IEEE, Los Alamitos, CA, 42–50. DOI: https://doi.org/10.1109/Correctness.2018.00010

[92] Yuan Yu, Tom Rodeheffer, and Wei Chen. 2005. RaceTrack: Efficient detection of data race conditions via adaptive tracking. *SIGOPS Oper. Syst. Rev.* 39, 5 (October 2005), 221–234. DOI: https://doi.org/10.1145/1095809.1095832

[93] T. Yuki, G. Gupta, D. Kim, T. Pathan, and S. Rajopadhye. 2012. Alphaz: A system for design space exploration in polyhedral model. In *Proceedings of the International Workshop on LCPC*. Springer, 17–31.

[94] M. Zheng, V. T. Ravi, F. Qin, and G. Agrawal. 2011. GRace: A low-overhead mechanism for detecting data races in GPU programs. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*. ACM, New York, NY, 135–146. DOI: https://doi.org/10.1145/1941553.1941574

[95] M. Zheng, V. T. Ravi, F. Qin, and G. Agrawal. 2014. GMRace: Detecting data races in GPU programs via a low-overhead scheme. *IEEE Trans. Parallel Distrib. Syst.* 25, 1 (2014), 104–115.

Received December 2019; revised July 2020; accepted August 2020