An Extended Polyhedral Model for SPMD Programs and its use in Static Data Race Detection

Prasanth Chatarasi, Jun Shirako, Martin Kong, and Vivek Sarkar

Rice University, Houston TX 77005, USA,
{prasanth,shirako,mkong,vsarkar}@rice.edu

Abstract. Despite its age, SPMD (Single Program Multiple Data) parallelism continues to be one of the most popular parallel execution models in use today, as exemplified by OpenMP for multicore systems and CUDA and OpenCL for accelerator systems. The basic idea behind the SPMD model, which makes it different from task-parallel models, is that all logical processors (worker threads) execute the same program with sequential code executed redundantly and parallel code executed cooperatively. In this paper, we extend the polyhedral model to enable analysis of explicitly parallel SPMD programs and provide a new approach for static detection of data races in SPMD programs using the extended polyhedral model. We evaluate our approach using 34 OpenMP programs from the `OmpSCR` and `PolyBench-ACC` [1] benchmark suites.

Keywords: SPMD parallelism; Data race detection; Polyhedral models; Phase mapping; Space mapping; May-Happen in Parallel Relation;

1 Introduction

It is widely recognized that computer systems anticipated in the 2020 time frame will be qualitatively different from current and past computer systems. Specifically, they will be built using homogeneous and heterogeneous many-core processors with 100's of cores per chip, and their performance will be driven by parallelism, and constrained by energy and data movement [21]. This trend towards ubiquitous parallelism has forced the need for improved productivity and scalability in parallel programming models. Historically, the most successful runtimes for shared memory multiprocessors have been based on bulk-synchronous Single Program Multiple Data (SPMD) execution models [10]. OpenMP [18] represents one such embodiment in which the programmer's view of the runtime is that of a fixed number of threads executing computations in "redundant" or "work-sharing" parallel modes.

As with other imperative parallel programming models, data races are a pernicious source of bugs in the SPMD model. Recent efforts on static data

[1] PolyBench-ACC derives from the PolyBench benchmark suite and provides OpenMP, OpenACC, CUDA, OpenCL and HMPP implementations.

race detection include approaches based on symbolic execution, e.g., [23,15], on polyhedral analysis frameworks, e.g., [3,24]. Past work on data race detection using polyhedral approaches have either focused on loop level parallelism, as exemplified by OpenMP's `parallel for` construct, or on task parallelism, as exemplified by X10's `async` and `finish` constructs, but not on general SPMD parallelism.

In this paper, we introduce a new approach for static detection of data races by extending the polyhedral model to enable analysis of explicitly parallel SPMD programs.[2] The key contributions of the paper are as follows:

1. An extension of the polyhedral model to represent SPMD programs.
2. Formalization of the May Happen in Parallel (MHP) relation in the extended polyhedral model.
3. An approach for static detection of data races in SPMD programs.
4. Demonstration of our approach on 34 OpenMP programs from the `OmpSCR` and the `PolyBench-ACC` OpenMP benchmark suites.

The rest of the paper is organized as follows. Section 2 summarizes the background for this work. Section 3 motivates the proposed approach for race detection with an example. Section 4 includes limitations of the existing polyhedral model, and the details of our extensions to the polyhedral model to represent SPMD programs. Section 5 shows how the MHP relation can be formalized in the extended model and describes our approach to compile-time data race detection. Section 6 contains our experimental results for data race detection. Finally, Section 7 summarizes related work, and Section 8 contains our conclusions and future work.

2 Background

This section briefly summarizes the SPMD execution model using OpenMP constructs as an exemplar, as well as an introduction to data race detection, that provides the motivation for our work. Then, we briefly summarize the polyhedral model since it provides the foundation for our proposed approach to static data race detection.

2.1 SPMD Parallelism using OpenMP

SPMD (Single Program Multiple Data) parallelism [9,10] continues to be one of the most popular parallel execution models in use today, as exemplified by OpenMP for multicore systems and CUDA, OpenCL for accelerator systems. The basic idea behind the SPMD model is that all logical processors (worker threads) execute the same program, with sequential code executed redundantly and parallel code (worksharing, barrier constructs, etc.) executed cooperatively.

[2] An earlier version of this paper was presented at the IMPACT'16 workshop [6], a forum that does not include formal proceedings.

In this paper, we focus on OpenMP [18] as an exemplar of SPMD parallelism. The OpenMP `parallel` construct indicates the creation of a fixed number of parallel worker threads to execute an SPMD parallel region. The OpenMP `barrier` construct specifies a barrier operation among all threads in the current `parallel` region. In this paper, we restrict our attention to textually aligned barriers, in which all threads encounter the same textual sequence of barriers. Each dynamic instance of the same `barrier` operation must be encountered by all threads, e.g., it is not permisible for a barrier in a then-clause of an if statement executed by (say) thread 0 to be matched with a barrier in an else-clause of the same if statement executed by thread 1. We plan to address textually unaligned barriers as part of the future work. However, many software developers believe that textually aligned barriers are better from a software engineering perspective.

The OpenMP `for` construct indicates that the immediately following loop can be parallelized and executed in a work-sharing mode by all the threads in the parallel SPMD region. An implicit barrier is performed immediately after a `for` loop, while the `nowait` clause disables this implicit barrier. Further, a `barrier` is not allowed to be used inside a `for` loop. When the `schedule(`*`kind`*`,` *`chunk_size`*`)` clause is attached to a `for` construct, its parallel iterations are grouped into batches of *chunk_size* iterations, which are then scheduled on the worker threads according to the policy specified by *kind*.

The OpenMP `master` construct indicates that the immediately following region of code is to be executed only by the master thread of the parallel SPMD region. Note that, there is no implied barrier associated with this construct.

2.2 Data Race Detection

Data races are a major source of semantic errors in shared memory parallel programs. In general, a data race occurs when two or more threads perform conflicting accesses (such that at least one access is a write) to a shared location without any synchronization among threads. Complicating matters, data races may occur only in some of the possible schedules of a parallel program, thereby making them notoriously hard to detect and reproduce. A large variety of static and dynamic data race detection techniques have been developed over the years with a wide range of guarantees with respect to the scope of the checking (schedule-specific, input-specific, or general) and precision (acceptable levels of false negatives and false positives) supported. Among these, the holy grail is static checking of parallel programs with no false negatives and minimal false positives. This level of static data race detection has remained an open problem for SPMD programs, even though there has been significant progress in recent years on race detection for restricted subsets of fork-join and OpenMP programs [16,23,15], as well as for higher-level programming models [3,24,4,2].

2.3 Polyhedral model

The polyhedral model is a flexible representation for arbitrarily nested loops [12]. Loop nests amenable to this algebraic representation are called *Static Control*

Parts (SCoP's) and represented in the SCoP format, which includes four elements for each statement, namely, iteration domains, access relations, dependence polyhedra/relations and the program schedule. In the original formulation of polyhedral frameworks, all array subscripts, loop bounds, and branch conditions in *analyzable* programs were required to be affine functions of loop index variables and global parameters. However, decades of research since then have led to a significant expansion of programs that can be considered analyzable by polyhedral frameworks [8].

Iteration domain, $\mathcal{D}^S$: A statement S enclosed by m loops is represented by an m-dimensional polytope, referred to as the iteration domain of the statement. Each point in the iteration domain is an execution instance $\boldsymbol{i} \in \mathcal{D}^S$ of the statement.

Access relation, $\mathcal{A}^S(\boldsymbol{i})$: Each array reference in a statement is expressed through an access relation, which maps a statement instance $\boldsymbol{i}$ to one or more array elements to be read/written. This mapping is expressed in the affine form of loop iterators and global parameters; a scalar variable is considered to be a degenerate (zero-dimensional) array.

Dependence relation, $\mathcal{D}^{S \to T}$: Program dependences in polyhedral frameworks are represented using dependence relations that map instances between two statement iteration domains, i.e., $\boldsymbol{i} \in S$ to $\boldsymbol{j} \in T$. These relations are then leveraged to compute a new program schedule that respects the order of the statement instances in the dependence.

Schedule, $\Theta^S(\boldsymbol{i})$**:** The execution order of a program is captured by the schedule, which maps instance $\boldsymbol{i}$ to a logical time-stamp. In general, a schedule is expressed as a multidimensional vector, and statement instances are executed according to the increasing lexicographic order of their timestamps.

3 Motivation

To motivate the proposed approach for static detection of data races, we discuss an explicitly parallel SPMD kernel as an illustrative example.

Illustrative Example. The example shown in Figure 1) is a 2-dimensional Jacobi computation from the `OmpSCR` benchmark suite [11]. The computation is parallelized using the OpenMP parallel construct with worksharing directives (lines 5, 11) and synchronization directives (implicit barriers from lines 5, 11). The first for-loop is parallelized (at line 5) to produce values of the array `uold`. Similarly, the second for-loop is parallelized (at line 11) to consume values of the array `uold`. The reduced `error` (from the reduction clause at line 11) is updated by only the `master` thread in the region (lines 26-29). Finally, the entire computation in lines 5–29 is repeated until it reaches the maximum number of iterations (or) the error is less than a threshold value. This pattern is very common in many stencil programs, often with multidimensional loops and multidimensional arrays. Although the worksharing parallel loops have implicit barriers, the programmer who contributed this code to the `OmpSCR` suite likely overlooked the

```
1 #pragma omp parallel private(resid, i)//tid-thread id
2 {
3     while (k <= maxit && error > tol) { //S1
4     /* copy new solution into old */
5 #pragma omp for
6       for (j=0; j<m; j++)
7         for (i=0; i<n; i++)
8           uold[i + m*j] = u[i + m*j];

10        /* compute stencil, residual and update */
11 #pragma omp for reduction(+:error)
12      for (j=1; j<m-1; j++)
13        for (i=1; i<n-1; i++) {
14          resid=(ax*(uold[i-1+m*j] + uold[i+1+m*j]) + ay*(uold[i+m*(j-1)]
15                    + uold[i+m*(j+1)]) + b*uold[i+m*j] - f[i+m*j]) / b;

17          /* update solution */
18          u[i + m*j] = uold[i + m*j] - omega * resid;

20          /* accumulate residual error */
21          error =error + resid*resid;
22        }

24      /* error check */
25 #pragma omp master
26      {
27        k++; //S2
28        error = sqrt(error) /(n*m); //S3
29      }
30   } /* while */
31 } /* end parallel */
```

Fig. 1. 2-D Jacobi kernel from `OmpSCR` benchmark suite

fact that a `master` region does not include a barrier. As a result, data races are possible in this example since statement S1's (at line 3) read access of variables `k, error` by a non-master thread can execute in parallel with an update of the same variables performed in statements S2 (at line 27) and S3 (at line 28) by the master thread. These races can be fixed by inserting another barrier immediately after the `master` region.

We observe that existing static race detection tools (e.g., [23,3]) are unable to identify such races since they don't model barriers inside of imperfectly nested sequential loops in the SPMD regions. We also observe that existing dynamic race detection tools such as Intel Inspector XE (2015 Update 1) in its `default` mode miss this true race and hybrid race detection tools such as ARCHER [2] incurred significant runtime overhead to detect this true race. In contrast, our proposed approach using the extended polyhedral model can identify such races at compile-time by effectively capturing execution phases from `barrier` directives via static analysis of SPMD regions.

4 Extended Polyhedral Model for SPMD Programs

In this section, we begin with discussing limitations of the polyhedral model for analyzing SPMD programs. Then, we summarize our extensions to the polyhedral model to support SPMD parallelism.

4.1 Limitations

The polyhedral model is an algebraic representation used to support compiler techniques for analysis and transformation of perfectly/imperfectly nested loops in sequential programs. Recent efforts [5] have extended polyhedral modeling techniques to explicitly parallel programs, but assuming the "serial-elision" property i.e., the property that removal of all parallel constructs results in a sequential program that is a valid (albeit inefficient) implementation of the parallel program semantics. Note that SPMD programs don't satisfy the "serial-elision" property because (for example) removing a barrier from an SPMD region would alter the semantics.

An interesting property of an explicitly parallel program is that it specifies a partial execution order unlike a sequential program, which specifies a total order. The schedule mapping (defined in Section 2.3) was originally introduced to represent the total order present in a sequential program. However, it can also be used to specify parallelism by assigning the same logical timestamp to multiple statement instances, thereby indicating that they can be executed at the same time. Still, this mapping is not sufficient to capture the partial order in a SPMD program. Hence, we extend the schedule with space and phase mappings (defined in the following sections) to explicitly capture the partial order.

4.2 Space (Allocation) mapping, $\Theta^S_A(i)$

Space (Allocation) mapping assigns a processor stamp to a statement instance that indicates a logical processor id on which the instance has to be executed. As a convenience for computing the space mapping, we 1) Replace the `omp parallel` region header by a logical parallel loop that iterates over threads, 2) Enclose the body of static scheduled worksharing loop in an `if` block with the condition on the thread iterator to be a function of lower and upper bounds, the loop chunk size and total number of threads participating in the worksharing loop (the last two are treated as fixed but unknown program parameters), 3) Insert an explicit `barrier` immediately after the worksharing loop (or) `single` region if a `nowait` clause is not specified, 4) Enclose the body of `master` region in an `if` block with the condition on the thread iterator to be zero. (Note that these transformations are only performed for the purpose of program analysis, and do not result in changes in the original program.) For the OpenMP program in Figure 1, the space mappings for statements S1, S2, and S3 are (tid), (0), and (0) respectively where `tid` is an iterator from the logical parallel loop (line 1).

4.3 Phase mapping, $\Theta^S_P(i)$

A key property of the SPMD programs is that their execution can be partitioned into a sequence of phases separated by barriers. It has been observed in past work that statements from different execution phases cannot execute concurrently [27]. Thus, only pairs of data accesses that execute within the same phase need to be considered as potential candidates for data races. The phase

mapping assigns a logical identifier, which we refer to as a phase stamp, to each statement instance. Thus, statement instances are executed according to increasing lexicographic order of their phase-stamps. We define reachable barriers for a statement instance 'S' as a set of barriers instances that can be executed after the statement instance 'S' without an intervening barrier instance. For the OpenMP program in Figure 1, reachable barriers for the statement `S2` are the implicit barrier instance (at line 9) in the next iteration of `while` loop and the implicit barrier at the end of the `parallel` region (at line 31). Finally, we compute the phase mapping for a statement instance as the union of the timestamps of all reachable barriers of the statement instance. There exists only one such reachable barrier at run-time under the assumption of textually aligned barriers and it would be one (based on the program parameters) from the statically determined set of reachable barriers.

Algorithm 1: Phase Mapping

Input : SCoP
Output: SCoP with phase mappings

begin

/* Extract initial schedules (time stamps) */

θ^S := Statement schedules from SCoP

θ^B := Barrier schedules from SCoP

/* Compute a map from statements to barriers such that elements of statements are lexicographically strictly smaller than those of barriers */

$\delta^{S \to B} := \{\boldsymbol{x} \to \boldsymbol{y} : \theta(\boldsymbol{x}) \prec \theta(\boldsymbol{y}), \boldsymbol{x} \in S, \boldsymbol{y} \in B\}$

/* Build a map from time stamps of statements to time stamps of barriers with lexicographically strictly smaller property */

$\delta^{\theta(S) \to \theta(B)} := (\theta^S)^{-1} \circ \delta^{S \to B} \circ \theta^B$

/* Extract a map from pairs of statement and barrier timestamps to their time difference */

$\delta^{(S,B) \to (\theta(B) - \theta(S))} := \{(\theta(\boldsymbol{x}) \to \theta(\boldsymbol{y})) \to (\theta(\boldsymbol{y}) - \theta(\boldsymbol{x})) : \boldsymbol{x} \in S, \boldsymbol{y} \in B\}$

/* Compute a map from each statement time stamp to the time stamp of the closest barrier instance for each lexical barrier */

$\beta := dom(lexmin(\delta^{(S,B) \to (\theta(B) - \theta(S))}))$

/* Compute a map (reachable barriers) from each statement instance to the closest barrier instance, among all lexical barriers */

$\beta^S := lexmin(\theta^S \circ \beta) \circ (\theta^B)^{-1}$

/* Compute phase mappings by union of timestamp of the reachable barrier instances to each statement instance */

Phase mappings, $\Theta^S_P := \beta^S \circ \theta^B$

end

Algorithm 1 summarizes the overall approach to compute the phase mappings for statements in a given SCoP. This approach can handle barriers within imperfectly nested sequential loops inside the SPMD region.

Initially, Lines 2-3 of the Algorithm 1 extract the schedules (time stamps) of both the regular statements and the barriers used. Then, Line 4 computes a map from the regular statements to barriers such that the elements of regular statements are lexicographically strictly smaller than those of barriers. Line 5 also computes another map from the time stamps of the regular statements to the time stamps of barriers, and keeping lexicographically strictly smaller property. Then, Line 6 essentially computes a set having difference of time stamps of regular statements and barriers; Line 7 computes the minimum of such differences and returns a map from the time stamps of regular statement to the time stamps of barrier having the minimum differences. This is then used in line 8 to determine the first/immediate barrier reachable for each regular statement instance. Finally, line 9 determines the phase mapping by assigning the timestamp of the first reachable barrier instance for a given statement instance.

5 Static Data Race Detection

In this section, we begin with our workflow for static data race detection. Then, we explain the formalization of MHP relations in the extended polyhedral model and describe our approach to compile-time data race detection.

5.1 Our workflow (PolyOMP)

The overall workflow is summarized in Figure 2, which is implemented as an extension to the Polyhedral Extraction Tool (PET) [22], and consists of the following components: 1) Conversion from input OpenMP-C program to Clang AST with the help of Clang-omp (version: 3.5) [7] and LLVM (version: 3.5.svn), 2) Conversion from Clang AST to PET AST (defined in [22]) (with the support for `omp parallel, for, parallel for, barrier, single, master` directives and nested parallel regions), 3) Extract SCoP from the PET AST, 4) Generate Extended SCoP (SCoP with space and phase mappings) from the PET AST and the SCoP, 5) Perform static race detection to detect races with the help of MHP relations from extended SCoP.

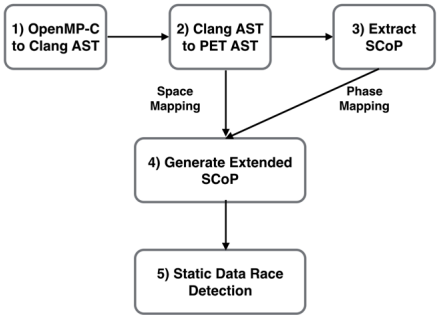

Fig. 2. Overview of PolyOMP

5.2 Computation of MHP Relations

MHP analysis statically determines if it is possible for execution instances of two statements (or the same statement) to execute in parallel [1]. In general, two statement instances S and T in a parallel region can execute in parallel iff both of them are in the same phase of computation (not ordered by synchronization) and are executed by different threads in the region. Algorithm 2 summarizes the overall steps to compute the MHP relations on a given pair of statements S, T in the SCoP. Non-affine and unknown functions appear in the space mapping (physical thread distribution) of statement instances while handling OpenMP worksharing loops with the static schedules, parametric chunk size and total number of threads. However, we conservatively compare name and arguments of space mapping of a thread with other threads to distinguish as two different threads.

Algorithm 2: MHP algorithm

Input : Statements S, T from the SCoP
Output: MHP Relations between S and T

begin

```
/* Extract space and phase mappings of statements S and T      */
```

$\Theta_A^S, \Theta_A^T :=$ Space mappings of S, T

$\Theta_P^S, \Theta_P^T :=$ Phase mappings of S, T

```
/* Compute a map from S to T such that they are in same phase  */
```

$\delta_{SamePhase}^{S \to T} := \Theta_P^S \circ (\Theta_P^T)^{-1}$

```
/* Compute a map from S to T such that they are on same thread */
```

$\delta_{SameThread}^{S \to T} := \Theta_A^S \circ (\Theta_A^T)^{-1}$

```
/* Compute the cross product of S and T                        */
```

$\delta_{CrossProduct}^{S \to T} := dom(\Theta_A^S) \times dom(\Theta_A^T)$

```
/* Compute a map from S to T such that they are run on different
   threads                                                     */
```

$\delta_{DiffThreads}^{S \to T} := \delta_{CrossProduct}^{S \to T}$ - $\delta_{SameThreads}^{S \to T}$

```
/* Compute MHP relations by intersecting the same phase and
   different thread maps of S and T (same map domain and range)
   */
```

$\delta_{MHP}^{S \to T} := \delta_{DiffThreads}^{S \to T} \cap \delta_{SamePhase}^{S \to T}$

end

5.3 Computation of Data Races

Detecting read-write and write-write data races becomes straightforward with the availability of MHP relations. In general, there exists a race between state-

ments S and T on memory location x iff MHP(S, T) is true, access relations of S and T intersect each other on memory location x and at-least one of them is a write. Our approach considers all possible pairs of the statements and builds data race conditions as per the above criteria and solves for the existence of solutions. Our approach is guaranteed to be exact (with neither false positives nor false negatives) if the input program satisfies all the standard preconditions of the polyhedral model (without any non-affine constructs). Thanks to the PET framework's [22] ability to handle non-affine constructs (in both data subscripts and control flow) elegantly in the form of may-write access relations, our approach now incorporates may-write access relations while computing data races and hence it may induce false positives (but not false negatives) for non-affine programs.

5.4 Limitations of PolyOMP

The current implementation supports OpenMP constructs such as `omp parallel for, parallel for, barrier, single, master` directives and nested parallel regions. Our tool currently does not perform pointer based analysis. However, any previous works on pointer analysis can be added as a pre-pass to our race detection stage. The support for analyzing SPMD programs with lock-based synchronization and task-based constructs are left as future work.

6 Experimental Evaluation

In this section, we present the evaluation of our approach for static race detection using the extended polyhedral model. Table 1 and Table 2 list the number of races discovered by our PolyOMP tool in the `OmpSCR` and `PolyBench-ACC` suites, along with the number of different `OpenMP` constructs in each benchmark.

6.1 OmpSCR Benchmarks suite

`OmpSCR`, an OpenMP Source Code Repository [11], is composed by a set of OpenMP applications written in C, C++ and Fortran. There are 18 OpenMP-C benchmarks in this repository, in which 6 benchmarks use `C structs` and pointer arithmetic. Since we defer support for `C structs` and pointer arithmetic to future work in our current tool chain, our results focus on the remaining 12 OpenMP-C benchmarks in `OmpSCR`, which are listed in Table 1. These results have been obtained on a quad core-i7 (2.2 GHz) machine with 16 GB main memory.

This benchmark suite contains known races, as reported in prior work on dynamic datarace detection in the ARCHER tool [2]. Our evaluation shows that PolyOMP is able to detect all of the documented races in the following applications using the static analysis algorithm in this paper: `Jacobi03, LoopA.bad, LoopB.bad1, LoopB.bad2`. All reported races (column `Reported`)

Table 1. Data races in subset of `OmpSCR` benchmark suite. Columns: Number of SPMD regions(#SPMD), Number of worksharing directives in a SPMD region(#WS), Number of barriers including implicit in a SPMD region (#Barriers), PolyOMP: race detection time, number of reported races, and number of false positives. ARCHER/ Intel Inspector XE (2015 Update 1 with `default` mode): Number of races reported.

Benchmark	#SPMD	#WS	#Barriers	PolyOMP			ARCHER	Intel Inspector XE
				Detection Time (Sec)	Reported	False+ves		
Jacobi01	2	1	1	1.38	2	2	0	0
Jacobi02	1	2	2	3.91	2	2	0	0
Jacobi03	1	3	3	1.54	4	2	2	0
Lud	1	1	1	0.30	0	0	0	1
LoopA.bad	1	1	1	0.20	1	0	1	2
LoopA.sol1	2	1	2	0.44	0	0	0	2
LoopA.sol2	1	0	2	1.21	7	7	0	0
LoopA.sol3	1	0	2	1.19	7	7	0	0
LoopB.bad1	1	1	1	0.20	1	0	1	2
LoopB.bad2	1	1	1	0.21	1	0	1	2
LoopB.pipe	1	0	2	2.40	7	7	0	0
C_pi	1	1	1	0.05	0	0	0	1
Total	**14**	**12**	**19**	**13.03**	**32**	**27**	**5**	**10**

were manually verified. (Note: each reported data race corresponds to a static pair of conflicting accesses). The `False +ves` column shows the number of reported races that actually are false positives. In addition, we compared our reported races with those reported by ARCHER[3]. Our tool computes races conservatively when unanalyzable control flow or data accesses are present and result in false positive races. This has been evident (27 false positives) in case of benchmarks `Jacobi01, Jacobi02, Jacobi03, LoopA.sol2, LoopA.sol3, LoopB.pipe` since these benchmarks contain linearized array subscripts. However, when the parallel region fully satisfies all the assumptions of standard polyhedral frameworks (e.g., all array accesses and branch conditions must be affine functions of the loop variables) then all reported races are true races. Even though Intel Inspector XE was able to identify some races, it has missed an important true race in case of `Jacobi03` (explained in Section 3) and it also reported additional FALSE ??races on iterators of parallel loops in case of `Lud, LoopA.bad, LoopA.sol1, LoopB.bad1, LoopB.bad2, C_pi` benchmarks.

6.2 PolyBench-ACC Benchmark Suite

We also use `PolyBench-ACC`, another benchmark suite partially derived from the standard PolyBench benchmark suite [13]. There are 32 OpenMP-C benchmarks in this suite, for which we were unable to compile 10 benchmarks due to compile-time errors arising from the usage of OpenMP directives in those codes. Thus, our results focus on the remaining 22 OpenMP-C benchmarks in `PolyBench-ACC`.

This benchmark suite is relatively new and is perhaps still in development compared to other benchmark suites. All of the benchmarks in this suite have

[3] ARCHER is known to not have any false positives or false negatives for a given input, but may have false negatives for inputs that it has not seen.

Table 2. Data races in subset of `PolyBench-ACC` (22) benchmark suite. Columns: Number of SPMD regions(#SPMD), Number of worksharing directives in a SPMD region(#WS), Number of barriers including implicit in a SPMD region (#Barriers), PolyOMP: race detection time, number of reported races, and number of false positives. Races reported by Intel Inspector XE tool (2015 Update 1 with `default` mode): Hang up (H), Application exception (A).

Benchmark	#SPMD	#WS	#Barriers	PolyOMP			Intel Inspector XE
				Detection Time (Sec)	Reported	False +ves	
Correlation	1	4	4	2.30	0	0	H
Covariance	1	3	3	1.04	0	0	H
2mm	1	2	2	0.64	0	0	0
3mm	1	3	3	1.13	0	0	0
Atax	1	2	2	0.37	2	0	2
Bicg	1	2	2	0.43	2	0	2
Cholesky	1	1	1	0.49	28	0	8
Doitgen	1	1	1	0.54	0	0	0
Gemm	1	1	1	0.34	0	0	0
Gemver	1	4	4	0.75	0	0	0
Gesummv	1	1	1	0.52	0	0	0
Mvt	1	2	2	0.32	0	0	0
Symm	1	1	1	0.64	5	0	5
Syrk	1	2	2	0.39	0	0	0
Syr2k	1	2	2	0.52	0	0	0
Trmm	1	1	1	0.28	1	0	1
Durbin	1	2	2	0.73	6	0	0
Gramschmidt	1	1	1	0.36	12	0	8
Lu	1	1	1	0.33	5	0	5
Convolution-2	1	1	1	0.25	0	0	0
Convolution-3	1	1	1	0.42	0	0	A
Fdtd-ampl	1	1	1	1.62	0	0	0
Total	**22**	**39**	**39**	**14.41**	**61**	**0**	**31**

statically analyzable control flow, affine subscripts and completely fit the assumptions of the polyhedral model without any conservative estimates. We manually verified the reported races and found the races to be real. It also verifies our claim that our approach is guaranteed to be exact (with neither false positives nor false negatives) if the input program satisfies all the standard preconditions of the polyhedral model (without any non-affine constructs).

Currently, we are not aware of any prior work reporting data races in this benchmark suite. Hence, we compared our reported races with those reported by the Intel Inspector XE tool (with *default* mode), which (unlike ARCHER) is known to have false negatives even for a given input. Overall, our tool reported a total of 61 races in this `PolyBench-ACC` benchmark suite out of which Intel Inspector XE could only find 31 races. The details are presented in Table 2. A table entry marked with the letter "H" indicates that the Intel Inspector XE tool would get into a hang mode for that benchmark, while a table entry marked with the letter "A" indicates that the Intel Inspector XE tool encountered an Application exception for that benchmark. The explanations for the races in the `PolyBench-ACC` benchmark suite are: 1) Majority of data races in `Cholesky, Gramschmidt` are on the non-privatized scalar variables in the worksharing loops, 2) Data races in `Atax, Bicg` are on the common array elements which are updated in a sequential loop of the SPMD region, 3) Remaining data

races in other benchmarks are because of dependences across the worksharing loops.

7 Related work

In this section, we discuss past work related to compile-time detection of data races, and the analysis of barriers in SPMD programs.

7.1 Static Race Detection

There is an extensive literature on identifying races in explicitly parallel programs (at compile-time [16,23,15,3,24,4], run-time [20], and hybrid combinations of both [19]). We focus our discussion on past work of static analysis techniques for identifying data races in SPMD-style parallel programs.

Symbolic approaches have received a lot of attention in analyzing parallel programs, mainly in the context of OpenMP. Yu et al's [23] work checks the consistency of multi-threaded programs with OpenMP directives using extended thread automata (with a tool called Pathg). However, their race detection is only guaranteed for a fixed number of worker threads. Ma et al. [15] use a symbolic execution-based approach (running the program on symbolic inputs and fixed number of threads) to detect data races in OpenMP codes, based on constraint solving using an SMT solver. The data races reported from this toolkit (called OAT) are applicable only to a fixed number of input threads, unlike our approach which allows the number of threads to be unknown.

Polyhedral based approaches have gained significant interest in analyzing parallel programs due to its ability to perform exact analysis on affine programs. Basupalli et al. [3] presented an approach (ompVerify) to detect data races inside a given worksharing loop using polyhedral dependence analysis. However, this approach handled only affine constructs and was limited to worksharing loops, rather than to general SPMD parallel regions. Yuki et al. [24] presented an adaptation of array data-flow analysis to X10 programs with finish/async parallelism. In this approach, the happens-before relations are first analyzed, and the data-flow is computed based on the partial order imposed by happen-before relations. This extended array dataflow analysis is used to certify determinacy in X10 finish/ async parallel programs by identifying the possibility of multiple sources of writes for a given read. Their extended work [25] formulated the happens-before relations with X10 clocks in a polyhedral context. This approach provides the race-free guarantee of clocked X10 programs by disproving all possible races. But, it doesn't provide races present in the input program since computing happens-before relations involves polynomials in a general case.

Atzeni et al. [2] introduced a hybrid static+dynamic approach (ARCHER) to achieve high accuracy, low overheads on large applications to detect data races. The static part of ARCHER tool leverages an existing polyhedral dependence analyzer to identify races in a given worksharing loop. Our static approach can

be complemented with the dynamic analysis of ARCHER tool to further reduce dynamic overheads as observed for the benchmark in Figure 1.

Among the verification techniques, Betts et al. [4] presented a technique for verifying race and divergence freedom of GPU kernels that are written in mainstream kernel programming languages such as OpenCL and CUDA. This tool (called GPUVerify) first translates a given GPU kernel into a sequential Boogie program that models the lock-step execution of two threads using a two-thread reduction method. The correctness of this sequential Boogie program implies race freedom of the original GPU kernel.

7.2 Barrier Analysis

The work by Yelick et al on concurrency analysis [14] computes MHP relations using a graph-based approach over single-valued expressions. Concurrent statements are identified using a depth-first search from a given statement. Their computed MHP information is conservative since it doesn't analyze statements and barriers at the instance level when they are enclosed in loops, in contrast to the exactness of our approach for affine programs. It also doesn't consider thread-mapping information when computing MHP.

Zhang et al. [26] proposed a barrier matching analysis for textually unaligned barriers, and introduced a phase partitioning concurrency algorithm in follow-on work [27]. While this approach can handle textually unaligned barriers for structurally correct programs, the papers do not discuss how to handle barriers nested in loops.

8 Conclusions & Future Work

This work is motivated by the observation that software with explicit parallelism is on the rise, and that SPMD parallelism is a common model for explicit parallelism as evidenced by the popularity of OpenMP, OpenCL and CUDA. As with other imperative parallel programming models, data races are a pernicious source of bugs in the SPMD model and may occur only in few of the possible schedules of a parallel program, thereby making them extremely hard to detect dynamically.

In this paper, we introduced a new approach for static detection of data races by extending the polyhedral model to enable analysis of explicitly parallel SPMD programs. We evaluated our technique using 34 OpenMP programs from the `OmpSCR` and `PolyBench-ACC` benchmark suites. We formalize the May Happen in Parallel (MHP) relations by adding "space" and "phase" dimensions to the schedule, and is guaranteed to be exact (with neither false positives nor false negatives) for identifying data races if the input program satisfies all the standard preconditions of the polyhedral model.

In summary, our contributions include the following: 1) An extension of the polyhedral model to represent SPMD programs, 2) Formalization of the May Happen in Parallel (MHP) relation in the extended model, 3) An approach

for static detection of data races in SPMD programs, and 4) Demonstration of our approach on 34 OpenMP programs from the `OmpSCR` and `PolyBench-ACC` benchmark suites.

As future work, we plan to leverage our framework to address problems such as redundant barrier optimization, detection of false sharing patterns, deadlock identification and coupling it with dynamic analysis techniques to prune false positives arising from unanalyzable data accesses, as done in [2,17].

References

1. Agarwal, S., Barik, R., Sarkar, V., Shyamasundar, R.K.: May-happen-in-parallel Analysis of X10 Programs. PPoPP, New York, NY, USA (2007)
2. Atzeni, S., Gopalakrishnan, G., Rakamarić, Z., Ahn, D.H., Laguna, I., Schulz, M., Lee, G.L., Protze, J., Müller, M.S.: Archer: Effectively Spotting Data Races in Large OpenMP Applications. In: IPDPS (2016)
3. Basupalli, V., Yuki, T., Rajopadhye, S., Morvan, A., Derrien, S., Quinton, P., Wonnacott, D.: ompVerify: Polyhedral Analysis for the OpenMP Programmer. IWOMP (2011)
4. Betts, A., Chong, N., Donaldson, A., Qadeer, S., Thomson, P.: GPUVerify: A Verifier for GPU Kernels. OOPSLA (2012)
5. Chatarasi, P., Shirako, J., Sarkar, V.: Polyhedral Optimizations of Explicitly Parallel Programs. In: PACT (2015)
6. Chatarasi, P., Shirako, J., Sarkar, V.: Static Data Race Detection for SPMD Programs Using an Extended Polyhedral Representation. In: IMPACT (2016)
7. CLANG OMP: CLANG Support for OpenMP 3.1. https://clang-omp.github.io
8. Collard, J.F., Barthou, D., Feautrier, P.: Fuzzy Array Dataflow Analysis. In: Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 92–101. PPOPP '95, ACM, New York, NY, USA (1995)
9. Cytron, R., Lipkis, J., Schonberg, E.: A Compiler-assisted Approach to SPMD Execution. Supercomputing, Los Alamitos, CA, USA (1990)
10. Darema, F., et al.: A Single-Program-Multiple-Data Computational model for EPEX/FORTRAN. Parallel Computing 7(1), 11–24 (1988)
11. Dorta, A.J., Rodriguez, C., Sande, F.d., Gonzalez-Escribano, A.: The OpenMP Source Code Repository. PDP, Washington, DC, USA (2005)
12. Feautrier, P., Lengauer, C.: Polyhedron Model. In: Padua, D.A. (ed.) Encyclopedia of Parallel Computing, pp. 1581–1592. Springer (2011)
13. Grauer-Gray, S., Xu, L., Searles, R., Ayalasomayajula, S., Cavazos, J.: Auto-tuning a High-Level Language Targeted to GPU Codes (2012)
14. Kamil, A., Yelick, K.: Concurrency Analysis for Parallel Programs with Textually Aligned Barriers. LCPC, Springer-Verlag, Berlin, Heidelberg (2006)
15. Ma, H., Diersen, S.R., Wang, L., Liao, C., Quinlan, D., Yang, Z.: Symbolic Analysis of Concurrency Errors in OpenMP Programs. ICPP, Washington, DC, USA (2013)
16. Mellor-Crummey, J.: Compile-time Support for Efficient Data Race Detection in Shared-memory Parallel Programs. PADD, New York, NY, USA (1993)
17. O'Callahan, R., Choi, J.D.: Hybrid Dynamic Data Race Detection. PPoPP (2003)
18. OpenMP Specifications. http://openmp.org/wp/openmp-specifications
19. Protze, J., Atzeni, S., Ahn, D.H., Schulz, M., Gopalakrishnan, G., Müller, M.S., Laguna, I., Rakamarić, Z., Lee, G.L.: Towards providing low-overhead data race detection for large OpenMP applications. In: Proceedings of the 2014 LLVM Compiler Infrastructure in HPC (2014)

20. Raman, R., Zhao, J., Sarkar, V., Vechev, M.T., Yahav, E.: Scalable and precise dynamic datarace detection for structured parallelism. In: PLDI (2012)
21. Sarkar, V., Harrod, W., Snavely, A.E.: Software Challenges in Extreme Scale Systems (Jan 2010), special Issue on Advanced Computing: The Roadmap to Exascale
22. Verdoolaege, S., Grosser, T.: Polyhedral Extraction Tool. Second International Workshop on Polyhedral Compilation Techniques (IMPACT 12), Paris, France
23. Yu, F., Yang, S.C., Wang, F., Chen, G.C., Chan, C.C.: Symbolic Consistency Checking of OpenMp Parallel Programs. LCTES (2012)
24. Yuki, T., Feautrier, P., Rajopadhye, S., Saraswat, V.: Array Dataflow Analysis for Polyhedral X10 Programs. PPoPP (2013)
25. Yuki, T., Feautrier, P., Rajopadhye, S.V., Saraswat, V.: Checking Race Freedom of Clocked X10 Programs. CoRR abs/1311.4305 (2013)
26. Zhang, Y., Duesterwald, E.: Barrier Matching for Programs with Textually Unaligned Barriers. PPoPP '07 (2007)
27. Zhang, Y., Duesterwald, E., Gao, G.: Concurrency Analysis for Shared Memory Programs with Textually Unaligned Barriers. In: LCPC (2008)