Static Data Race Detection for SPMD Programs via an Extended Polyhedral Representation

Prasanth Chatarasi, Jun Shirako, and Vivek Sarkar
Department of Computer Science, Rice University
{prasanth,shirako,vsarkar}@rice.edu

ABSTRACT

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 is that all logical processors (worker threads) execute the same program, with sequential code executed redundantly and parallel code executed cooperatively. As with other imperative parallel programming models, data races are a pernicious source of bugs in the SPMD model. While there have been some recent advances in techniques for dynamic detection of data races for SPMD and other parallel programming models, there is a notable lack of tools for static detection of data races in SPMD programs.

The polyhedral model is a powerful algebraic framework that has enabled significant advances to static analysis and transformation of sequential affine (sub)programs, relative to traditional AST-based approaches. In this paper, we introduce a new approach for static detection of data races by extending the polyhedral model to enable static analysis of explicitly parallel SPMD programs. 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 by generating race constraints that can be solved by an SMT solver such as Z3, and 4) Demonstration of our approach by automatic generation of race constraints from two sample OpenMP programs. Further, 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 (no non-affine constructs).

Keywords

SPMD Parallelism, Static Data Race Detection, Polyhedral Model, Z3 SMT Solver

IMPACT 2016
Sixth International Workshop on Polyhedral Compilation Techniques
Jan 18-20, 2015, Prague, Czech Republic
In conjunction with HiPEAC 2016.

http://impact.gforge.inria.fr/impact2016

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, their performance will be driven by parallelism, and constrained by energy and data movement [24]. 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 [15]. OpenMP [21] 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 race detection include approaches based on symbolic execution, e.g., [29, 18], and on polyhedral analysis frameworks, e.g., [3, 30]. Past work on data race detection using polyhedral approaches have either focused on loop 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 static analysis of SPMD programs[1]. The key contributions of the paper are as follows:

- An extension of the polyhedral model to represent SPMD programs.
- Formalization of the May Happen in Parallel (MHP) relation in the extended model.
- An approach for static detection of data races in SPMD programs by generating race constraints that can be solved by an SMT solver such as Z3.
- Demonstration of our approach by automatic generation of race constraints from two sample OpenMP programs.

[1]A two-page summary abstract of this approach was presented at the PACT ACM SRC'15 poster session [7].

The rest of the paper is organized as follows. Section 2 summarizes the background for this work. Section 3 includes an overview of our proposed approach and discusses two examples to explain our approach. Section 4 discusses the details of our extensions to the polyhedral model to represent SPMD programs, and shows how the MHP relation can be formalized in the extended model. Section 5 describes our approach to compile-time data race detection via the extended polyhedral framework, and illustrates our approach by automatic generation of race constraints from two sample OpenMP programs that can be submitted to the Z3 solver. Finally, Section 6 and Section 7 summarize related work, as well as our conclusions.

2. BACKGROUND

This section briefly summarizes the SPMD execution model using OpenMP constructs as an exemplar, as well as past work on data race detection, that together provide the motivation for our work. We also briefly summarize the polyhedral model and the Z3 SMT solver since they contribute to the foundation for our proposed approach to static data race detection.

2.1 SPMD Parallelism using OpenMP

SPMD (Single Program Multiple Data) parallelism [14, 15] 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 is that all logical processors (worker threads) execute the same program, with sequential code executed redundantly and parallel code (worksharing constructs, barriers, etc.) executed cooperatively.

In this paper, we focus on OpenMP [21] 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 number of threads can be specified in the code, or in an environment variable (`OMP_NUM_THREADS`), or via a runtime function, `set_omp_num_threads()` that is called before the `parallel` region starts execution.

The OpenMP `barrier` construct specifies a barrier operation among all threads in the current `parallel` region. Each dynamic instance of the same `barrier` operation must be encountered by all threads, e.g., it is not permitted 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.

The `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 scheduled on the worker threads according to the policy specified by *kind*.

In this paper, we restrict our attention to OpenMP parallel loops with *kind* = `dynamic` and *chunk_size* = 1, which implies that each iteration can be executed by any thread in the `parallel` region. The justification for this assumption is that if a program is proved to be data-race-free with this assumption, it is guaranteed to be data-race-free for all schedule clauses. Thus, this is a reasonable assumption to make when debugging a parallel program.

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 variable 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. So, static data race detection remains open even though there has been significant progress in recent years on static data race detection for restricted subsets of fork-join and OpenMP programs [19, 29, 18], as well as for higher-level programming models [3, 30, 4], there has been a notable lack of attention paid to static data race detection for general SPMD programs.

2.3 Polyhedral model

In this section, we summarize polyhedral representations of sequential programs. The polyhedral model is a flexible representation for arbitrarily nested loops. Loop nests amenable to this algebraic representation are called *Static Control Parts* (SCoPs) and represented in the SCoP format, where includes three elements for each statement, namely, iteration domain, access relations, and 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 great expansion of programs that can be considered analyzable by polyhedral frameworks [11, 12, 13].

Iteration domain, $\mathcal{D}^S$: A statement S enclosed by m loops is represented by an m-dimensional polytope, referred to as an iteration domain of the statement [17]. Each element in the iteration domain of the statement is regarded as a statement instance $\vec{i} \in \mathcal{D}^S$.

Access relation, $\mathcal{A}^S(\vec{i})$: Each array reference in a statement is expressed through an access relation, which maps a statement instance $\vec{i}$ to one or more array elements to be read/written [28]. 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.

Schedule, $\Theta^S(\vec{i})$: The sequential execution order of a program is captured by the schedule, which maps instance $\vec{i}$ 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 time-stamps.

Scattering function: In the context of polyhedral frameworks, the term "scattering function" is often used to combine a schedule $\Theta^S(\vec{i})$ with further information related to parallel execution e.g., execution order/location/processor [2, 10]. For example, Parallel execution can be modeled by schedules in which multiple statement instances are assigned

the same time stamp.

Most existing polyhedral modeling techniques assume that the input program is sequential and do not model any explicit parallelism that may be present in the input program. In recent work [8], we addressed the problem of analyzing and transforming programs with explicit parallelism (doall and task parallelism in OpenMP 4.0) , that satisfy the *serial-elision* property by extending existing polyhedral modeling techniques. In contrast, this work focuses on extending the polyhedral model to enable static datarace detection in SPMD programs, which, in general, do not satisfy the serial elision property.

2.4 Z3 SMT Solver

Z3 is a state of the art theorem prover and SMT solver from Microsoft Research [16] that is used to check the satisfiability of logical formulae. The output from the solver can be *sat/ un-sat/ un-dec*: the input logical formula is satisfiable (there exists an assignment that marks logical formula as true)/ unsatisfiable/ undecidable. The Z3 solver is best used as a component in the context of other tools that require solving logical formulae and exposes many API facilities to make it convenient for tools to map into Z3. It has support for uninterpreted functions, non-linear arithmetic, divisions, bit vectors operations, recursive datatypes, and quantifiers. Examples of satisfiability queries submitted to the Z3 solver and responses from it, are presented in Appendix A.

3. OVERVIEW OF OUR APPROACH

Our approach considers a read/ write or write/write pair on the same shared variable in the same parallel region to be a data race, and generates race constraints accordingly. This race constraint encodes the necessary conditions for conflicting accesses to that shared variable by two threads. These race constraints are fed unto the Z3 SMT solver to check for the existence of solutions.

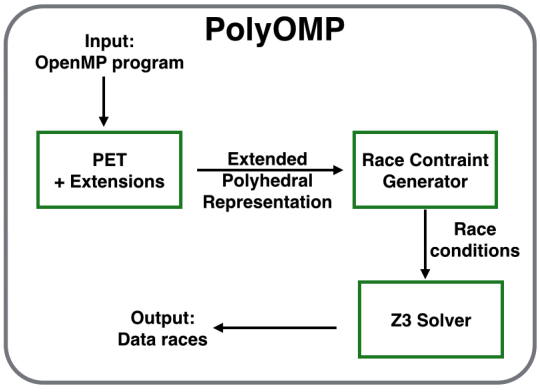

Figure 1: Overview of our approach

The overall approach is summarized in Figure 1, and consists of the following components: 1) Conversion from input OpenMP-C code to an extended polyhedral representation. This step involves generating PET data structures [27, 26] from the Clang AST and converting the PET data structures to an extended polyhedral representation (SCoP), 2) Race constraint generator to generate race conditions, 3) Z3 SMT solver to compute a solution to the race conditions (if one exists). Currently, the implementations of generating race constraints from the extended polyhedral representation (Component 2) and integrating with the Z3 solver (Component 3) are still work in progress. To explain the proposed approach, we discuss two simple SPMD examples containing worksharing constructs and barrier directives that represent common patterns found in OpenMP programs. We intentionally inserted data races in these examples, to illustrate how they are detected by our approach.

3.1 Example 1

Listing 1: Example of a race between statements inside different worksharing constructs

```
1 // tid - Thread id
2 // T - Total number of threads
3 #pragma omp parallel  shared(A) {
4  #pragma omp for schedule(dynamic,1) nowait
5     for(int i = 0; i < N; i++) {
6         S1:  A[i] =  ...  ;
7     }
8
9  #pragma omp for schedule(dynamic,1)
10    for(int j = 0; j < N; j++) {
11        S2:    ... =  A[j];
12    }
13 }
```

The parallel region in Listing 1 spans the block of code from line 3 to line 13. The first for-loop is parallelized (at line 4) to produce values of the array `A`. Similarly, the second for-loop is parallelized (at line 9) to consume values of the array `A`. This pattern is very common in many application programs, often with multidimensional loops and multidimensional arrays (e.g., see the `applu` application in the SPEC OMP benchmark suite). The problem in this example arises from the fact that the first loop contains a `nowait` clause. There are many plausible reasons why the `nowait` clause may have been inserted by the programmer. For example, it's possible that the second loop did not originally contain an access of `A[j]` when the `nowait` clause was inserted in the first loop. As a result, the later addition of a read access to `A[i]` in the second loop leads to a data race between the thread that produces an element of array `A` in line 6 and the thread that consumes that element in line 11. Another reason may be that the code was originally written with static schedule clauses, and later changed to dynamic scheduled clauses for improved load balance (if, say, the statements S1 and S2 have non-uniform execution times). (Note that this data race is possible because of the `dynamic` clause; there would not be a data race if both loops had `static` clauses instead.)

We observe that existing polyhedral based race detection tools, such as ompVerify, are unable to identify such races since they concentrate on races among statements within a given parallel for-loop [3]. Our race detection tool identifies such races by using our extended polyhedral framework to generate constraints for each pair of read/write or write/write accesses on the same shared variable in the same parallel region. For example, the race constraint between the write of `A[i]` at line 6 and the read of `A[j]` at line 11 is

generated as follows:

$$(0 \leq i < N) \wedge (0 \leq j < N) \wedge (i = j) \wedge (0 \leq tid_i < T)$$
$$\wedge (0 \leq tid_j < T) \wedge (tid_i \neq tid_j) \wedge (0 = 0)$$

where T, tid_i and tid_j represent the total number of threads, the thread id that executes iteration '`i`' in the first for-loop and thread id that executes iteration '`j`' in the second for-loop respectively. This modeling of dynamic schedules in worksharing loops and of the `nowait` clause are discussed in Section 4 and Section 5.

3.2 Example 2

Listing 2: Example of a race between statements within a sequential loop nest and separated by a barrier

```
1 // tid - Thread id
2 // T - Total number of threads
3 #pragma omp parallel  shared(A) {
4     for(int i = 0; i < N; i++) {
5         for(int j = 0; j < N; j++) {
6             S1: int temp = A[tid + i + j];
7             #pragma omp barrier
8             S2: A[tid] += temp;
9         }
10    }
11 }
```

Listing 2 shows another parallel region (from lines 3-11), which consists of a sequential doubly nested loop enclosing a set of statements (at lines 6, 8) and a barrier (at line 7) between them. This pattern is commonly used in accelerator programming where each thread proceeds in a lock step fashion. An example to in which arrays are accessed using the thread identifier can be seen in the `kmeans` application in the Rodinia benchmark suite [9]. Line 6 reads data to a temporary variable from the shared array `A` with the index of `tid+i+j` and Line 8 reads the data from the temporary variable to update the shared array `A` with the index of `tid`. Note that `tid` represents the thread identifier, which can be obtained via the OpenMP runtime library `omp_get_thread_num()`. Although the programmer provided an explicit barrier to prevent a race in a given i,j-iteration across multiple threads between statement S1 reading from `A[tid+i+j]` and statement S2 writing to `A[tid]`, the data race in this example is due to the lack of synchronizations across the j-iterations; thereby the S1's read access of `A[tid+i+j]` on the next j-iteration can execute in parallel with an update of the same location (`A[tid]`) performed in statement S2 by another thread. This bug can be fixed by inserting another barrier after statement S2. Existing static race detection tools for OpenMP are unable to identify such races arising from barriers nested within loops, because doing so requires static analysis to detect which statement instances in nested loop structures are synchronized/ordered via such synchronization constructs [29]. Our modeling of barriers is discussed in Section 4.

4. POLYHEDRAL EXTENSIONS TO SPMD PROGRAMS

4.1 Extended Polyhedral Representation

The scattering function is a key data structure present in the SCoP intermediate representation used in polyhedral frameworks. It is used to describe the order in which statement instances have to be executed relative to each other, and contains a schedule map of the form $\Theta^S(\vec{i})$, which assigns logical timestamps to the statement instances $S(\vec{i})$. These timestamps indicate the logical time at which different statement instances should be executed (in lexicographic increasing order of timestamps). These logical timestamps can be multidimensional (in the case of multidimensional schedules) to simplify the representation of ordering information among the statements in a sequential program.

A major difference between a sequential program and an explicitly parallel program (such as OpenMP) is that a sequential program specifies a total execution order among statement instances, and an explicitly parallel program specifies a partial execution order. The existing schedule function ($\Theta^S(\vec{i})$) for sequential programs captures the total execution order very effectively. The same schedule function ($\Theta^S(\vec{i})$) can also specify parallelism by assigning the same logical timestamp to multiple statement instances, thereby indicating that they can be executed at the same time. But, this representation is not sufficient to specify the kinds of parallelism and synchronization constructs present in SPMD parallel programs (e.g., barriers, point-to-point synchronizations). In this paper, the scattering function of a statement is extended with additional mapping information such as allocation (*space*) and computational phase (*phase*) to capture the precise semantics of SPMD OpenMP constructs. We refer to this kind of scattering function as a *(space/ phase/ time)* mapping. The semantics of this mapping is explained below.

4.2 Allocation (space) mapping, $\Theta^S_A(\vec{i})$

Allocation (space) mapping assigns processor stamps that indicate the logical processor on which a statement instance $S(\vec{i})$ has to be executed. Algorithm 1 shows[2] the overall approach to compute space mapping for a given SPMD-style OpenMP parallel region with support for regular statements, worksharing constructs and barriers, while deferring support for other OpenMP constructs to future work. We replace the parallel region header by a logical parallel loop that iterates over threads (with the iterator `tid`) as a convenience for computing spaces. The iteration vector of a statement outside worksharing regions explicitly contains `tid`, e.g., $\vec{i} = (tid, i, j)$ in Listing 2, while statements within worksharing regions have regular iteration vectors.

In the case of a worksharing `for-loop` with a dynamic schedule, the mapping function between the loop iterator and processor is not known at compile time. So, we represent its processor mapping with an existential quantifier. On the other hand with static schedule, the mapping function is non-affine (with the chunk size and number of threads as parameters); as mentioned earlier, we defer support for static schedules to future work. The space mappings for statements (S1, S2) in the OpenMP program in Listing 1 are as follows:

$$\{[\Theta^{S1}_A(i_{\vec{S1}})] \rightarrow [t1] : \text{ exists } x : 0 \leq x < T \text{ and } t1 == x\}$$
$$\{[\Theta^{S2}_A(i_{\vec{S2}})] \rightarrow [t2] : \text{ exists } x : 0 \leq x < T \text{ and } t2 == x\}$$

[2]We chose to describe this mapping as an algorithm rather a formal definition, for ease of presentation.

Algorithm 1 Algorithm to compute space mapping

Input:
R, S: A given SPMD-style OpenMP parallel region R with a set of statements, S.
Output:
$\Theta_A^{S_i}(\vec{i})$, mapping from statement instances to logical processors (worker threads).
Method:
for each statement S_i in S **do**
  if S_i is part of worksharing `for-loop` **then**
    { $[\Theta_A^{S_i}(\vec{i_S})] \rightarrow$ [t] : exists x: $0 \leq x < T$ and t == x } (in isl-notation)
    /* T is symbolic parameter that represents the total number of threads. The `exists` clause captures the semantics of the `dynamic` schedule clause, by indicating that any thread can be chosen. We defer support for the `static` schedule clause to future work. */
  else
    { $[\Theta_A^{S_i}(\vec{i_S})] \rightarrow$ [tid] } (in isl-notation)
    /* `tid` refers to the "current" SPMD thread.*/
  end if
end for

4.3 Phase mapping, $\Theta_P^S(\vec{i})$

A key property of 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 [31]. 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, that we refer to as a *phase stamp*, to each statement instance $S(\vec{i})$. Thus, statement instances are executed according to increasing lexicographic order of their phase-stamps. Algorithm 2 summarizes the overall algorithm to compute the phase mapping in a given SPMD-style OpenMP parallel region, which can include barriers within sequential perfectly nested loops. (We defer computing phases for barriers enclosed within imperfect loop nests to future work.) Phase stamps can be multidimensional, just like timestamps in schedules.

Reaching barriers for a statement 'S' is defined as a set of barriers (including barrier instances if barriers are inside loops) that can be executed after the statement 'S' without an intervening barrier. Reaching barriers of the statement are computed from control flow graph (CFG) of the input program, in similar to reaching definitions of a variable. Then, the phase mapping for a statement is computed as OR of the time stamps for the reaching barriers of that statement, along with reachability conditions from the statement to the barriers. The phase mappings for statements (S1, S2) in the OpenMP program in Listing 2 are as follows.

$$\begin{aligned}\Theta_P^{S1}(\vec{i}) &= (i, j),\ \vec{i} = (i, j)\\ \Theta_P^{S2}(\vec{i}) &= (i, j+1) \text{ if } j < N-1\\ &\quad\ (i+1, 0) \text{ if } j = N-1\end{aligned}$$

In case of the statement S2 (i,j), there are two instances of the barrier (line 7) that can be reachable from the statement S2. The time stamp of the first barrier instance is (i, j+1) and it can be reachable if the value of `j` is less than N-1.

Algorithm 2 Algorithm to compute phase mapping

Input:
R, S: A given SPMD-style OpenMP parallel region R with a set of statements as S.
Output:
$\Theta_P^{S_i}(\vec{i})$, mapping from statement instances to phase stamp.
Method:
Preprocessing:
Assume input program is sequential (being run on single thread), compute schedules Θ^S for all the statements (considering barriers also as regular statements) in S
for each statement S_i in S **do**
  Compute reaching barriers (B_{S_i}) for the statement S_i based on control flow graph (CFG) of input program
  for each reaching barrier (b) in B_{S_i} **do**
    $\Theta_P^{S_i}(\vec{i}) = \vee(\Theta^b$ with reachability conditions between S_i and b)
  end for
end for

The time stamp of the second barrier instance is (i+1, 0) and it can be reachable if the value of `j` is N-1, which means that this barrier is reachable in the first iteration of j-loop in the next iteration of i-loop.

4.4 Extended Access Relation

Another major difference between a sequential program and an explicitly parallel program (such as OpenMP) is the presence of data sharing attributes. In an explicitly parallel program, array data variables can be declared with different sharing attributes, such as `shared, private, reduction,` etc. These data sharing attributes play an important role in the analysis of explicitly parallel programs, and are incorporated into *access relations* for the analysis. Variables declared within `parallel` regions are also considered to be `private` variables.

4.5 May Happen in Parallel Relation, MHP

Parallel programming languages offer many high-level parallel constructs to create parallel regions and synchronize threads. All these parallel constructs indicate the relative progress and interactions of threads during execution. Further, these interactions among threads can impact the possible execution order of statement instances. For example, statements before and after a `barrier` are ordered within a region as they cannot be executed simultaneously. Knowledge of these possible orderings can be very helpful when debugging parallel programs.

May-Happen-in-Parallel (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 be run in parallel if and only if both of them are in same phase of computation (not ordered by synchronization) and are executed by different threads in the region. MHP(S, T) = true iff $(\Theta_A^S(\vec{i_S}) \neq \Theta_A^T(\vec{i_T})) \wedge (\Theta_P^A(\vec{i_S}) = \Theta_P^T(\vec{i_T}))$. This condition appears quite simple because MHP contains less information than the Happens Before (HB) relation. If MHP(S, T) is true, then we know that HB(S, T) and HB(T,

S) must both be false. However, if MHP(S, T) is false, then we know that one of HB(S, T) and HB(T, S) must both be true and the other false, but there is insufficient information in MHP(S, T) to indicate which is which.

5. DATA RACE DETECTION ALGORITHM FOR SPMD PROGRAMS

We identify data races in SPMD-style parallel programs by taking advantage of extra semantic information provided by the user via explicit parallel constructs/ directives in the program. Our approach relies on the phase/ space mapping introduced in Section 4 for May-Happen-in-Parallel (MHP) analysis. The scope of applicability has limitations due to the current restriction on the barrier timestamp modeling and its assumption of a perfect loop nets, which will be addressed in future work.

5.1 Algorithm

Algorithm 3 shows the overall approach to identify a race between a given pair of read/ write or write/ write accesses on the same shared variable in a given SPMD region.

Algorithm 3 Data race detection algorithm

1: **Input:**
2: R, A, f_R, S, $\vec{i_S}$, $\mathcal{D}^S$: Read R in statement S. S reads from a shared array (or scalar) A with access function f_R and iteration vector $\vec{i_S} \in \mathcal{D}^S$
3: W, A, f_W, T, $\vec{i_T}$, $\mathcal{D}^T$: Write W in statement T. T writes to a shared array (or scalar) A with access function f_W and iteration vector $\vec{i_T} \in \mathcal{D}^T$.
4: **Method:**
5: Generate race constraint as below:
$(\vec{i_S} \in \mathcal{D}^S) \wedge (\vec{i_T} \in \mathcal{D}^T) \wedge (f_R = f_W) \wedge$ (MHP(S, T) = true), where MHP(S, T) is defined as: $(\Theta^S_A(\vec{i_S}) \neq \Theta^T_A(\vec{i_T})) \wedge (\Theta^A_P(\vec{i_S}) = \Theta^T_P(\vec{i_T}))$.
6: Submit the race constraint to the Z3 SMT solver [16] to check for the existence of solutions
7: **Output:**
8: If the race constraint is not satisfiable, there will not be a race between the accesses f_R in statement S and f_W in statement T.
9: If the race constraint is satisfiable, there may be a race between accesses f_R in statement S and f_W in statement T if conservative estimations (for non-affine constructs) are used in the representation and analysis such as may-access relations. Otherwise, the identified race is accurate.
10: If the race constraint is undecidable, we still report it as a potentially false positive race between accesses f_R in statement S and f_W in statement T.

Our race detection algorithm considers a read/write or write/write pair on the same shared variable in the same parallel region and generates the race constraint accordingly. This race constraint encodes necessary conditions for conflicting accesses to that shared variable by two threads. Then, the race constraint is passed onto the Z3 SMT solver to check for the existence of solutions. If the race constraint is not satisfiable, then we conclude that there are no races on those pair of accesses. 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) and the race constraints are decidable by the Z3 SMT solver. If the conservative estimations (for non-affine constructs) are used during representation and analysis such as may-access relations, then this approach may induce false positives. With the restrictions over the type of parallel constructs supported in this paper, the race conditions are restricted to Presburger formulae. In this case, integer mathematical libraries such as isl and Omega can be used to provide the required answer. In future, we plan to consider more general parallel constructs in SPMD regions such as worksharing loops with static schedules, critical sections, etc. In these parallel constructs, the MHP conditions will not always be Presburger formulae. In order to support such non-affine relations in future, we will need to use the generality of the Z3 SMT solver to check for the satisfiability of the race conditions. In the rest of section, we illustrate the data race detection algorithm using the examples introduced in Section 3.

5.2 Race detection for Example 1

The iteration domains, scattering functions and access relations of the statements (S1, S2) in the OpenMP program in Listing 1 are as follows:

- Statement S1:
  - Iteration vector, $\vec{i_{S1}}$ = (i)
  - Domain, $\mathcal{D}^{S1}$ = { i | 0 ≤ i < N }
  - Access function, $\mathcal{A}^{S1}(\vec{i_{S1}})$ = (A[i])
  - Space/ Phase/ Time mapping,
    - $\{[\Theta^{S1}_A(\vec{i_{S1}})] \rightarrow [tid_{S1}] :$ exists $x : 0 \leq x < T$ and $tid_{S1} == x\}$,
    - $\Theta^{S1}_P(\vec{i_{S1}})$ = (0),
    - $\Theta^{S1}(\vec{i_{S1}})$ = (0, i)
- Statement S2:
  - Iteration vector, $\vec{i_{S2}}$ = (j)
  - Domain, $\mathcal{D}^{S2}$ = { j | 0 ≤ j < N }
  - Access function, $\mathcal{A}^{S2}(\vec{i_{S2}})$ = (A[j])
  - Space/ Phase/ Time mapping,
    - $\{[\Theta^{S2}_A(\vec{i_{S2}})] \rightarrow [tid_{S2}] :$ exists $x : 0 \leq x < T$ and $tid_{S2} == x\}$,
    - $\Theta^{S2}_P(\vec{i_{S2}})$ = (0),
    - $\Theta^{S2}(\vec{i_{S2}})$ = (1, j)

The race constraint between the write of `A[i]` at line 6 (S1) and the read of `A[j]` at line 11 (S2) is generated as follows:

$$(0 \leq i < N) \wedge (0 \leq j < N) \wedge (i = j) \wedge (0 \leq tid_{S1} < T)$$
$$\wedge (0 \leq tid_{S2} < T) \wedge (tid_{S1} \neq tid_{S2}) \wedge (0 = 0)$$

The condition $(tid_{S1} \neq tid_{S2})$ comes from the fact that the space dimension of these statements should be different i.e, S1 and S2 should be executed by two different threads. The implicit barriers at the end of all `parallel for` loops are modeled like any other barrier.The presence of the nowait

clause ensures that all statement instances in this example execute in the same phase (phase 0), so the condition (0 = 0) is added to race constraint. If there was not a `nowait` clause (which implies that S1 and S2 execute in different phases), the condition (0 = 1) would have been added to the race constraint, thereby ensuring that the condition is unsatisfiable. With the nowait clause, the race condition shown above (with the 0 = 0 term) is fed into the Z3 SMT solver as a satisfiability query and the Z3 returns a valid assignment (i = 0, N = 1, tid_{S1} = 0, j = 0, tid_{S2} = 1, T = 2) to the constraint (explained in Appendix A). This assignment clearly indicates a race between iteration instance (i = 0) of statement S1 and iteration instance (j = 0) of statement S2. This race is precise as it doesn't involve any conservative estimates based on non-affine constructs.

5.3 Race detection for Example 2

The iteration domains, scattering functions and access relations for the statements S1 and S2 in the OpenMP program in Listing 2 are as follows:

- Statement S1:
  - Iteration vector, $\vec{i_{S1}} = (\text{tid}_{S1}, \text{i}, \text{j})$
  - Domain, $\mathcal{D}^{S1} = \{ (\text{i}, \text{j}) \mid 0 \leq \text{i}, \text{j} < \text{N} \}$
  - Access function, $\mathcal{A}^{S1}(\vec{i_{S1}}) = (\text{A}[\text{tid}_{S1} + \text{i} + \text{j}])$
  - Space/ Phase/ Time mapping,
    - $\{ [\Theta_A^{S1}(\vec{i_{S1}})] \rightarrow [\text{tid}_{S1}] \}$
    - $\Theta_P^{S1}(\vec{i_{S1}}) = (\text{i}, \text{j})$
    - $\Theta^{S1}(\vec{i_{S1}}) = (0, \text{i}, 0, \text{j}, 0)$
- Statement S2:
  - Iteration vector, $\vec{i_{S2}} = (\text{tid}_{S2}, \text{i}', \text{j}')$
  - Domain, $\mathcal{D}^{S2} = \{ (\text{i}', \text{j}') \mid 0 \leq \text{i}', \text{j}' < \text{N} \}$
  - Access function, $\mathcal{A}^{S2}(\vec{i_{S2}}) = (\text{A}[\text{tid}_{S2}])$
  - Space/ Phase/ Time mapping,
    - $\{ [\Theta_A^{S2}(\vec{i_{S2}})] \rightarrow [\text{tid}_{S2}] \}$
    - $$\Theta_P^{S2}(\vec{i_{S2}}) = \begin{array}{l} (i', j' + 1) \text{ if } j' < N - 1 \\ (i' + 1, 0) \text{ if } j' = N - 1 \end{array}$$
    - $\Theta^{S2}(\vec{i_{S2}}) = (0, \text{i}', 0, \text{j}', 1)$

The race constraint between the write of `A[tid]` at line 8 (S2) and the read of `A[tid + i + j]` at line 6 (S1) is generated as follows:

$$\begin{aligned} &(0 \leq i < N) \wedge (0 \leq j < N) \wedge (0 \leq i' < N) \wedge (0 \leq j' < N) \\ &\wedge (tid_{S1} + i + j = tid_{S2}) \wedge (tid_{S1} \neq tid_{S2}) \\ &\wedge ((i = i' + 1 \wedge j = 0 \wedge j' = N - 1) \\ &\quad \vee (i = i' \wedge j = j' + 1 \wedge j' < N - 1)) \end{aligned}$$

These race constraints are fed into the Z3 SMT solver as a satisfiability query and the Z3 returns a valid assignment (N=2, i = 1, j = 1, tid_{S1} = 0, i′ = 0, j′ = 1, tid_{S2} = 2) to the constraints (explained in Appendix A). This assignment clearly indicates a race between the statement S1 and S2 across iterations of `i,j-loop`. This race is precise as it doesn't involve any conservative estimates based on non-affine constructs. We note that recent work on symbolic analysis of OpenMP programs [29, 18] would not be able to identify such data races since they do not analyze the computational phases present in the SPMD model that underlies OpenMP programs.

6. RELATED WORK

There is an extensive body of literature on identifying races in explicitly parallel programs (at compile-time [19, 29, 18, 3, 30, 4], run-time [23], and hybrid combinations of both [22]). We focus our discussion on past work that is most closely related to static analysis techniques for identifying data races in SPMD-style parallel programs.

Among the static analysis techniques, symbolic approaches have received a lot of attention in analyzing parallel programs, especially in the context of OpenMP. Yu et al. [29] presented a symbolic approach for checking consistency of multi-threaded programs with OpenMP directives using extended thread automata (with a tool called Pathg). This approach uses the *Omega* library for constraint solving and a symbolic simulator for guided witness search for consistency. However, their checking is only guaranteed for a fixed number of worker threads, whereas our approach generates race constraints based on may-happen-in-parallel relations for phases in SPMD computations and effectively checks for data races for all values of numbers of worker threads. Ma et al. [18] also use a symbolic execution-based approach (running the program on symbolic inputs and fixed number of threads) to detect data races and deadlocks in OpenMP codes, based on constraint solving using an SMT solver. This tool (called OAT) focuses on a variety of OpenMP directives such as worksharing, barriers and locks. The data races and deadlocks reported from this toolkit is applicable only to a fixed number of input threads unlike our approach which takes number of threads as variable. Thus, a key difference between our approach and those based on symbolic execution, is that we extended the polyhedral framework to generate constraints that can be applicable to variable number of worker threads.

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 race freedom of the original kernel is implied by the correctness of the sequential Boogie program. The tool proves the correctness of this program by using existing modular techniques for program verification. On the other hand, our approach identify races in the input parallel SPMD kernel by modeling into the extended polyhedral representation without converting into its sequential Boogie program.

Recent race detection work of Basupalli et.al [3] is based on polyhedral modeling to identify races in OpenMP parallel `for-loops` and is integrated into the standard open source Eclipse IDE. This tool automatically converts a region of code into polyhedral representation, and analyzes for data races using polyhedral analysis techniques. However, this tool is limited to `omp parallel for` constructs with worksharing loops, whereas our approach is applicable to parallel SPMD programs in general.

A number of papers addressed the problem of data-flow analysis of explicitly parallel programs, including adapta-

tion of array data-flow analysis to X10 programs with finish/async parallelism [30]. 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 write for a given read. Their work [30] does not encompass analysis of SPMD programs (which would have been possible if their work also supported X10 programs with clocks).

In our past work [8], we addressed the problem of analyzing and transforming programs with explicit parallelism (doall and task parallelism in OpenMP 4.0) that satisfy the *serial-elision* property, i.e., the property that removal of all parallel constructs results in a sequential program that is a valid implementation of the parallel program semantics. That work starts by enabling a conservative dependence analysis of a given region of code, which may contain non-affine constructs. Next, it identifies happens-before relations from the explicitly parallel constructs, such as tasks and parallel loops, and intersects them with the conservative dependences. Finally, the resulting set of dependences is passed on to a polyhedral optimizer, such as PLuTo [5, 6] or PolyAST [25], to enable transformation of explicitly parallel programs with unanalyzable data accesses. However, the approach in [8] does not apply to general SPMD parallel programs (as in OpenMP parallel regions), since they do not satisfy the serial-elision property.

7. 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. Complicating matters, data races may occur only in few of the possible schedules of a parallel program, thereby making them extremely hard to detect dynamically. However, effective approaches to static data race detection remains an open problem, despite significant progress in recent years.

In this paper, we introduced an extension of the polyhedral model to represent SPMD programs and formalized the May Happen in Parallel (MHP) relation in this extended model, by adding "space" and "phase" dimensions to the scattering function. We also provided an approach for static detection of data races in SPMD programs by generating race constraints that can be solved by an SMT solver such as Z3. 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).

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 by generating race constraints that can be solved by an SMT solver such as Z3, and 4) Demonstration of our approach by automatic generation of race constraints from two sample OpenMP programs.

As part of future work, we would like to combine the proposed static race detection with dynamic analysis to prune false positives arising from conservative analysis, as has been done with hybrid data race detection for Java program [20]. At first, the static analysis can be applied to remove non-racy accesses. Then, dynamic analysis can be performed to take care of the rest. This hybrid approach can reduce false positives from static analysis and reduce unnecessary runtime overhead incurred by the dynamic analysis.

8. REFERENCES

[1] S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar. May-happen-in-parallel Analysis of X10 Programs. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 183–193, New York, NY, USA, 2007. ACM.

[2] C. Bastoul. Code Generation in the Polyhedral Model Is Easier Than You Think. *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, 0:7–16, 2004.

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

[4] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson. GPUVerify: A Verifier for GPU Kernels. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 113–132, New York, NY, USA, 2012. ACM.

[5] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic Transformations for Communication-minimized Parallelization and Locality Optimization in the Polyhedral Model. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*, CC'08/ETAPS'08, pages 132–146, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[7] P. Chatarasi and V. Sarkar. Extending Polyhedral Model for Analysis and Transformations of OpenMP programs. In *Microsoft ACM Student Research Competition (SRC) at PACT-2015*, San Francisco, CA, USA, 2015.

[8] P. Chatarasi, J. Shirako, and V. Sarkar. Polyhedral Optimizations of Explicitly Parallel Programs. In *Proc. of The 24th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, San Francisco, CA, USA, 2015.

[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54,

Washington, DC, USA, 2009. IEEE Computer Society.

[10] CLooG: The chunky loop generator. http://www.cloog.org.

[11] J.-F. Collard, D. Barthou, and P. Feautrier. Fuzzy Array Dataflow Analysis. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 92–101, New York, NY, USA, 1995. ACM.

[12] B. Creusillet and F. Irigoin. Interprocedural Array Region Analyses. *Int. J. Parallel Program.*, 24(6):513–546, Dec. 1996.

[13] B. Creusillet and F. Irigoin. Exact versus approximate array region analyses. In *Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '96, pages 86–100, London, UK, UK, 1997. Springer-Verlag.

[14] R. Cytron, J. Lipkis, and E. Schonberg. A Compiler-assisted Approach to SPMD Execution. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, Supercomputing '90, pages 398–406, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[15] F. Darema et al. A Single-Program-Multiple-Data Computational model for EPEX/FORTRAN. *Parallel Computing*, 7(1):11–24, 1988.

[16] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[17] P. Feautrier and C. Lengauer. Polyhedron model. In D. A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1581–1592. Springer, 2011.

[18] H. Ma, S. R. Diersen, L. Wang, C. Liao, D. Quinlan, and Z. Yang. Symbolic Analysis of Concurrency Errors in OpenMP Programs. In *Proceedings of the 2013 42Nd International Conference on Parallel Processing*, ICPP '13, pages 510–516, Washington, DC, USA, 2013. IEEE Computer Society.

[19] J. Mellor-Crummey. Compile-time Support for Efficient Data Race Detection in Shared-memory Parallel Programs. In *Proceedings of the 1993 ACM/ONR Workshop on Parallel and Distributed Debugging*, PADD '93, pages 129–139, New York, NY, USA, 1993. ACM.

[20] R. O'Callahan and J.-D. Choi. Hybrid Dynamic Data Race Detection. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, pages 167–178, New York, NY, USA, 2003. ACM.

[21] OpenMP Specifications. http://openmp.org/wp/openmp-specifications.

[22] J. Protze, S. Atzeni, D. H. Ahn, M. Schulz, G. Gopalakrishnan, M. S. Müller, I. Laguna, Z. Rakamarić, and G. L. Lee. Towards providing low-overhead data race detection for large OpenMP applications. In *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC*, pages 40–47. IEEE Press, 2014.

[23] R. Raman, J. Zhao, V. Sarkar, M. T. Vechev, and E. Yahav. Scalable and precise dynamic datarace detection for structured parallelism. In J. Vitek, H. Lin, and F. Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 531–542. ACM, 2012.

[24] V. Sarkar, W. Harrod, and A. E. Snavely. Software Challenges in Extreme Scale Systems. January 2010. Special Issue on Advanced Computing: The Roadmap to Exascale.

[25] J. Shirako, L.-N. Pouchet, and V. Sarkar. Oil and Water Can Mix: An Integration of Polyhedral and AST-based Transformations. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '14, 2014.

[26] S. Verdoolaege. isl: An Integer Set Library for the Polyhedral Model. In K. Fukuda, J. Hoeven, M. Joswig, and N. Takayama, editors, *Mathematical Software âĂŞ ICMS 2010*, volume 6327 of *Lecture Notes in Computer Science*, pages 299–302. Springer Berlin Heidelberg, 2010.

[27] S. Verdoolaege and T. Grosser. Polyhedral extraction tool. *Second International Workshop on Polyhedral Compilation Techniques (IMPACT 12), Paris, France*, 2012.

[28] D. G. Wonnacott. *Constraint-based Array Dependence Analysis*. PhD thesis, College Park, MD, USA, 1995. UMI Order No. GAX96-22167.

[29] F. Yu, S.-C. Yang, F. Wang, G.-C. Chen, and C.-C. Chan. Symbolic Consistency Checking of OpenMp Parallel Programs. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES '12, pages 139–148, New York, NY, USA, 2012. ACM.

[30] T. Yuki, P. Feautrier, S. Rajopadhye, and V. Saraswat. Array Dataflow Analysis for Polyhedral X10 Programs. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, 2013.

[31] Y. Zhang, E. Duesterwald, and G. Gao. Concurrency Analysis for Shared Memory Programs with Textually Unaligned Barriers. In V. Adve, M. GarzarÃąn, and P. Petersen, editors, *Languages and Compilers for Parallel Computing*, volume 5234 of *Lecture Notes in Computer Science*, pages 95–109. Springer Berlin Heidelberg, 2008.

APPENDIX

A. Z3 SMT SOLVER

A.1 SMT logical formulae for example 1

The race constraint between the write of `A[i]` at line 6 (S1) and the read of `A[j]` at line 11 (S2) is generated as follows:

$$(0 \le i < N) \wedge (0 \le j < N) \wedge (i = j) \wedge (0 \le tid_{S1} < T)$$
$$\wedge\ (0 \le tid_{S2} < T) \wedge (tid_{S1} \ne tid_{S2}) \wedge (0 = 0)$$

The equivalent Z3 query looks as follows and can be run in http://rise4fun.com/z3.

```
(declare-const N Int)
(declare-const i Int)
(declare-const j Int)
(declare-const T Int)
(declare-const tid_s1 Int)
(declare-const tid_s2 Int)
(push)
(assert (>= i 0))
(assert (< i N))
(assert (>= j 0))
(assert (< j N))
(assert (= i j))
(assert (>= tid_s1 0))
(assert (< tid_s1 T))
(assert (>= tid_s2 0))
(assert (< tid_s2 T))
(assert (not( = tid_s1 tid_s2)))
(check-sat)
(get-model)
```

The output from the Z3 solver is as follows.

```
sat
(model
  (define-fun i () Int
    0)
  (define-fun N () Int
    1)
  (define-fun tid_s1 () Int
    0)
  (define-fun j () Int
    0)
  (define-fun tid_s2 () Int
    1)
  (define-fun T () Int
    2)
)
```

One satisfiable assignment to the race constraint is (i = 0, N = 1, tid_{S1} = 0, j = 0, tid_{S2} = 1, T = 2).

A.2 SMT logical formulae for example 2

The race constraint between the write of `A[tid]` at line 8 (S2) and the read of `A[tid + i + j]` at line 6 (S1) is generated as follows:

$$(0 \le i < N) \wedge (0 \le j < N) \wedge (0 \le i' < N) \wedge (0 \le j' < N)$$
$$\wedge\ (tid_{S1} + i + j = tid_{S2}) \wedge (tid_{S1} \ne tid_{S2})$$
$$\wedge\ ((i = i' + 1 \wedge j = 0 \wedge j' = N - 1)$$
$$\vee\ (i = i' \wedge j = j' + 1 \wedge j' < N - 1))$$

The equivalent Z3 query looks as follows and can be run in http://rise4fun.com/z3.

```
(declare-const N Int)
(declare-const i Int)
(declare-const j Int)
(declare-const i1 Int)
(declare-const j1 Int)
(declare-const T Int)
(declare-const tid_s1 Int)
(declare-const tid_s2 Int)
(push)
(assert (>= i 0))
(assert (< i N))
(assert (>= j 0))
(assert (< j N))
(assert (>= i1 0))
(assert (< i1 N))
(assert (>= j1 0))
(assert (< j1 N))
(assert (= i j))
(assert (= (+ (+ tid_s1 i) j) tid_s2))
(assert (not( = tid_s1 tid_s2)))
(assert (or
  (and (and (= i i1) (= j (+ j1 1)) )
            (< j1 (- N 1)))
  (and (and (= i (+ i1 1)) (= j j1) )
            (= j1 (- N 1)))))
(check-sat)
(get-model)
```

The output from the Z3 solver is as follows.

```
sat
(model
  (define-fun i () Int
    1)
  (define-fun tid_s1 () Int
    0)
  (define-fun tid_s2 () Int
    2)
  (define-fun j () Int
    1)
  (define-fun N () Int
    2)
  (define-fun i1 () Int
    0)
  (define-fun j1 () Int
    1)
)
```

One satisfiable assignment to the race constraint is (N=2, i = 1, j = 1, tid_{S1} = 0, i′ = 0, j′ = 1, tid_{S2} = 2)