OpenSHMEM Checker - A Clang Based Static Checker for OpenSHMEM

Md Abdullah Shahneous Bari*, Ujjwal Arora*, Varun Hegde*, Tony Curtis*, and Barbara Chapman*†
{MdAbdullah.ShahneousBari, Ujjwal.Arora, Varun.Hegde, Anthony.Curtis, Barbara.Chapman}@stonybrook.edu
*Stony Brook University, †Brookhaven National Laboratory

***Abstract*—Compilers are generally not aware of the semantics of library-based parallel programming models such as MPI and OpenSHMEM, and hence are unable to detect programming errors related to their use. To alleviate this issue, we developed a custom static checker for OpenSHMEM programs based on LLVM's Clang Static Analyzer framework (CSA). We leverage the Symbolic Execution engine of the core Static Analyzer framework and its path-sensitive analysis to check for bugs on all OpenSHMEM program paths. We have identified common programming mistakes in OpenSHMEM programs that are detectable at compile-time and provided checks for them in the analyzer. They cover: utilization of the right type of memory (private vs. symmetric memory); safe/synchronized access to program data in the presence of asynchronous, one-sided communication; and double-free of memories allocated using OpenSHMEM memory allocation routines. Our experimental analysis showed that the static checker successfully detects bugs in OpenSHMEM code.**

***Index Terms*—Static Checker, Static Analysis, OpenSHMEM, Compiler Analysis**

I. Introduction

Parallel programming is complex; writing an error-free, portable, and performant parallel program can be a Herculean task for application developers. Compilers play a vital role in detecting errors in application programs. However, many existing parallel programming models (e.g., MPI, OpenSHMEM) are library-based, often supporting multiple programming languages (e.g., C/C++, Fortran) but lacking specialized compiler support for error detection. As a result, the burden of error detection with respect to the program's parallelization falls on the application developers and requires them to be extra vigilant to avoid bugs in their program. Reliance on runtime error detection can be difficult and may also result in undetected bugs in production code. Thus researchers use static and dynamic tools [1]–[3] to detect compile-time and runtime bugs when possible. Yet most existing tools are focused on MPI due to its popularity in the HPC community.

Other programming models with a growing user-base do not have the same tool support. OpenSHMEM is one such programming model. It is a Partitioned Global Address Space (PGAS) programming model which can be beneficial for computations with an irregular communication pattern (e.g., graph-based applications). Its performance advantages are due to a reliance on asynchronous, one-sided communications along with RMA (Remote Memory Access) support in recent hardware and its careful implementation. Yet the features (e.g., asynchronous, one-sided communication) that provide some of its key benefits lead to challenges with respect to writing an error-free OpenSHMEM-based parallel program.

To address this issue, we designed and developed a custom static checker, 'OpenSHMEM Checker' for OpenSHMEM programs based on LLVM. LLVMs Clang static analyzer (CSA) provides a framework that can be used to build custom, domain-specific checks for C/C++ programs. It offers two different techniques that can be used to create them: 1) AST-based checks, which utilize the information provided by the Clang Abstract Syntax Tree (AST), and, 2) Path sensitive checks, which can leverage the Symbolic Execution engine, which explores all paths in a program via path-sensitive analysis. We developed the OpenSHMEM checker based on its path-sensitive analysis to provide a comprehensive and accurate error detection mechanism. The checker will analyze a program with respect to its usage of OpenSHMEM routines and provide compile-time errors, warnings, and suggestions related to them. We designed it in such a way that it can be adapted for other library-based PGAS programming models with minimal effort.

In order to perform this work, we first identified common programming mistakes in the OpenSHMEM programming model. Based on this investigation, we decided to implement checks for three different errors that are frequent in OpenSHMEM code and amenable to static detection. Those are:

- Check that inspects whether a call to an OpenSHMEM routine conforms to the specification
- Check for double free in OpenSHMEM-specific allocations (symmetric heap allocation/de-allocation)
- Experimental support for data race detection

II. OpenSHMEM and Common Programming Mistakes

A. OpenSHMEM

The OpenSHMEM Specification [4], [5] defines a library-based PGAS (Partitioned Global Address Space) programming model/interface for C and C++. It uses a Single Program Multiple Data (SPMD) approach to provide local- and global-views of program data, split across communicating processes on 1 or more compute nodes. Like other PGAS models (e.g., GASNet, Global Arrays, UPC) OpenSHMEM takes advantage of network capabilities such as Remote Direct Memory Access (RDMA) [6] to allow efficient data movement that is

978-1-6654-3281-8/21/$31.00 ©2021 IEEE

decoupled from synchronization. OpenSHMEM is an open-source community effort to develop a software ecosystem for the scientific community and hardware vendors to ensure portability. There is a number of open-source implementations, e.g. OSSS-UCX, Ohio State University, Sandia National Lab, Oak Ridge National Lab, Open-MPI; and some from vendors, e.g. HPE/Cray, NVIDIA/Mellanox, IBM.

The OpenSHMEM API defines a library interface with routines to satisfy the communication needs of parallel applications. Those of most relevance to this paper include: point-to-point RDMA and atomic memory operations (AMO); and collective memory management, communication, and synchronization operations. Listing 1 shows an example of a skeleton OpenSHMEM program that performs halo exchange. OpenSHMEM routines are painted red. It starts by initializing the OpenSHMEM library and necessary resources using `shmem_init` (analogous to `MPI_Init` in MPI) and finishes by releasing all resources used by the OpenSHMEM library using `shmem_finalize`. In between, other OpenSHMEM routines are used to perform collective memory management (`shmem_malloc`, `shmem_free`), point-to-point RDMA (`shmem_put`), and synchronization (`shmem_barrier_all`).

```
1  #include <shmem.h>
2  ...
3  int main()
4  {
5    shmem_init(); //Initialize OpenSHMEM Library
6    ...
7    int* privMem = (int*) malloc(...);
8    int* symMem = (int*) shmem_malloc(...);
9    for(timeStep=0; timeStep<MAX_TIMESTEP; timeStep++)
10   {
11     do_some_computation();
12     //Halo exchange to neighbor, One-sided write
13     shmem_put(symMem, privMem, numElem, neighborPE);
14     shmem_barrier_all(); //Global barrier
15   }
16   ...
17   free(privMem);
18   shmem_free(symMem);
19   shmem_finalize(); //Finalize OpenSHMEM Library
20   return 0;
21 }
```

Listing 1: OpenSHMEM program performing halo exchange

1) OpenSHMEM Memory Model: OpenSHMEM programs consist of processes (Processing Elements, or PEs; analogous to MPI ranks) that communicate using point-to-point or collective operations. Data in the PEs can be marked as "symmetric", meaning it is exposed to the communication layer between PEs (typically an inter-connect such as Infiniband [7], or shared memory in a node) and can be read/written directly by other PEs. Infiniband and other networks enable native one-sided communication in hardware that frees the application or OS from dealing with progress issues.

2) Remote Memory Access Routines: Two core OpenSHMEM RDMA routines are the generic forms `shmem_put` and `shmem_get`[1], which allow a PE to respectively write to, or read from, the symmetric memory of another PE.

"Put" routines can allow for highly asynchronous low overhead access to another PE's symmetric memory, which can be exploited by applications with irregular/sparse communication patterns [8]. However, this asynchrony means that "put" operations do not guarantee completion when they return. So the application must provide later-synchronization to ensure consistency when data is needed, with the most common method being a global barrier.

3) Synchronization and Ordering Routines: OpenSHMEM synchronization and ordering/completion is discussed below:

`shmem_barrier, shmem_barrier_all` Provide collective synchronization over a subset of PEs and all PEs respectively.

`shmem_quiet` The PE calling quiet ensures remote completion of remote access operations and stores to symmetric data objects.

`shmem_fence` The PE calling fence ensures ordering of Put, AMO, and memory store operations to symmetric data objects with respect to a specific destination PE.

4) Collective Communication Routines: OpenSHMEM provides collective routines for Broadcast, Collection, Reduction, All-to-All, synchronization/ barrier, and symmetric memory management. All or subsets of PEs determined by the team (analogous to communicator in MPI) can participate in the collective operations.

OpenSHMEM was designed to enable high performance by exploiting the support for Remote Direct Memory Access (RDMA) available in modern network interconnects. It allows for highly efficient data transfers without incurring the software overhead that comes with message-passing communication. However, enabling those features required OpenSHMEM to introduce concepts such as symmetric/private memory, allocation/de-allocation of those symmetric memories, safe/unsafe access to those memories, which are not native to programming languages used with OpenSHMEM. As a result, programming errors resulting from these concepts are not detectable by a generic C/C++ compiler such as LLVM.

B. Common Programming Mistakes in OpenSHMEM

In this section we provide an overview of the most common errors in OpenSHMEM through representative examples. We use some core OpenSHMEM features and routines (described in Table I) to explain these programming errors.

1) Violation of OpenSHMEM Semantics- Using Wrong Kind of Memory: One of the most common mistakes observed in OpenSHMEM programming is using the wrong kind of memory. OpenSHMEM has the concept of private and symmetric memory and violating their usage results in undefined behavior. Listing 2 shows an example of such a case. In line 5, call to `shmem_put` requires a symmetric memory buffer of a remote process as the first argument (described in Table I); however, in this example, a private memory buffer 'privMem' is used instead. Using private memory 'privMem' where a symmetric memory is expected would result in undefined behavior (and worse yet, termination of the program depending on the implementation).

[1]Explicit typed versions also exist for basic C types such as "int", "short", "float". The full list is in the specification [5]

TABLE I: Description of some basic OpenSHMEM concepts and routines

Concept	Description
Symmetric memory	Potion of the memory that can be directly accessed by remote processes.
Private memory	Portion of the memory that is only accessible to the process that owns it. Generic concept of memory found in C/C++.
void shmem_init ()	Initializes the OpenSHMEM library. It is a collective operation that all PEs must call before any other OpenSHMEM routine may be called. Analogous to MPI_Init.
void shmem_finalize ()	Finalizes the OpenSHMEM library by releasing all resources used by the library. Analogous to MPI_Finalize.
void shmem_put (TYPE *dest, const TYPE *source, size_t nelems, int pe)	Writes 'nelems' amount of data of type 'TYPE' from 'source' buffer to remote process no. 'pe' s symmetric memory 'dest'. The routines return after the data has been copied out of the source array on the local PE. The delivery of the data is not guaranteed upon return from this routine. Further synchronization routines must be called to guarantee the delivery of the data.
void shmem_get (TYPE *dest, const TYPE *source, size_t nelems, int pe)	Copies/reads 'nelems' amount of data of type 'TYPE' from the symmetric memory 'source' of the remote process no. 'pe' to local memory buffer 'dest'. The routines return after the data has been delivered to the dest array on the local PE.
void *shmem_malloc (size_t size)	Allocates symmetric memory in the heap. Analogous to generic C routine 'malloc' except it allocates symmetric memory.
void shmem_free (void *ptr)	Releases/frees previously allocated symmetric memory. Analogous to generic C routine 'free' except it frees symmetric memory.

Unfortunately, a generic compiler (e.g., LLVM) would not be able to detect this error. Since C/C++ does not have the concept of private and symmetric memory, it can not distinguish between them. Therefore, it is up to the programmer to make sure the correct type of memory is used or risk having a buggy code with undefined behavior at runtime.

2) Double-free of Heap Allocated Symmetric Memory: Trying to free an already freed memory is a common programming mistake in C/C++. Some compilers add extra checks (e.g., LLVM has a dedicated static checker for this) to catch this error. However, OpenSHMEM has a specific type of double-free error that the generic double-free checkers can not catch. It stems from the OpenSHMEM specific symmetric memory allocation/deallocation routines. OpenSHMEM provides routines (e.g., `shmem_malloc`, `shmem_free`) to allocate and free heap-allocated symmetric memory. However, these symmetric memory allocation/free routines are OpenSHMEM-specific; therefore, the double-free error arising for symmetric memory is not caught by generic double-free checkers. Extra support must be added to catch this specific type of double-free error in OpenSHMEM.

3) Unsafe/Unsynchronized Access to Program Data: OpenSHMEM provides the means for asynchronous, one-sided communication by allowing access to remote process' (PE) symmetric memory without that process' acknowledgment. Due to this asynchronous communication nature, OpenSHMEM has a relaxed memory consistency; in other words, the completion of a communication routine usually has to be confirmed by synchronization routines. Therefore, non-blocking accesses to a PE's symmetric memory between two synchronization points may result in an inconsistent state of that PE's symmetric memory. We use the code example shown in Listing 3 to explain this in the context of an OpenSHMEM program. Here we use `shmem_put` in line 7 to write to the remote PE no '1's symmetric memory 'symMem'. However, returning from `shmem_put` does not guarantee the completion of actual write in the 'symMem' in remote process 'PE 1'; it just confirms that the data has been copied out of source array 'src' on the local PE. The actual write in the remote PE may happen at any time in the future. Only a synchronization routine (e.g., `shmem_barrier_all`) after `shmem_put` can guarantee the completion. Therefore, when we use `shmem_get` in line 12 to access (read from) the remote process 'PE 1's symmetric memory 'symMem' that we wrote to previously using `shmem_put`, we can not guarantee whether 'symMem' would have the old value (before `shmem_put`), or new value (after `shmem_put`), since we don't know if `shmem_put` was able to complete the write to 'symMem'. Hence, we may have a data race condition. Therefore, read from 'symMem' is unsafe without a synchronization routine before it.

4) Potential Performance Degradation Due to Excessive Synchronization: While synchronization constructs are one of the main building blocks of OpenSHMEM and an absolute necessity for writing a correct OpenSHMEM program as explained in the previous section, using too many and unnecessary synchronization could heavily degrade the application performance. However, application developers tend to overuse synchronization constructs in their programs in their quest for correctness. Although this does not affect the correctness of the program but may degrade performance heavily, hence finding cases of excessive synchronization is extremely important.

5) Deadlock Due to Missing Synchronization: OpenSHMEM has the concept of collective routines like other parallel programming models (e.g., MPI collectives) that are executed by all processes or a subset of processes executing the program. However, if a process that is supposed to participate in a collective doesn't do so, it may lead to a deadlock scenario.

```
1   ...
2   int *privMem = (int *)malloc(...); //private mem
3   //Error: Expects a symmetric memory
4   //Not detectable by generic compiler
5   shmem_put(privMem, &src, 1, pe);
6   ...
```

Listing 2: Using wrong memory type (symmetric vs. private)

```
1   ...
2   //Allocate symmetric memory
3   int *symMem = (int *)shmem_malloc(...);
4   // Write to symMem of PE no. '1', Write is not
5   // guaranteed to complete without proper
6   // synchronization
7   shmem_put(symMem, src, 1, 1);
8   //Reading from the symMem of PE no. '1'
9   //Missing synchronization construct
10  //Error: NOT safe to access 'symMem' variable
11  //Possible race condition
12  shmem_get(src, symMem, 1, 1);
13  ...
```

Listing 3: Unsafe/unsynchronized access to program data

C. OpenSHMEM and MPI

The Message Passing Interface (MPI) is widely used to write parallel programs in the HPC community. Although, traditionally it utilizes a two-sided communication approach, it provides semantics for one-sided communication as well. Since, MPI is well known across the HPC community, in Table II we provide a comparison of the programming mistakes prevalent in MPI and OpenSHMEM.

III. Clang Static Analyzer

A compiler gathers substantial information during compilation. The Clang compiler, which is part of the LLVM compiler suite, allows external tools to utilize this information for various purposes (e.g., the 'clang-tidy' tool uses it for diagnosing and fixing typical programming errors).

The 'Clang Static Analyzer (CSA)' is one such tool that tries to find defects in a program by symbolically executing (imaginary execution, as if reading the source code and imagining what would happen if it was run) it without actually running it.

To achieve this, the analyzer uses algebraic symbols (with constraints and bounds) to denote unknown values (e.g., variable values that are dependent on input and only available at runtime). During the symbolic execution process [9], it uses these symbols (and their bounds) and the Clang CFG (Control Flow Graph) to build a *graph of reachable states*, called an *Exploded Graph*. The *Exploded Graph* consists of all feasible paths through the CFG that were found by the analyzer. Each node in the graph represents pairs of program states and program points. The program state represents the abstract state of the program (e.g., mapping from source code locations to values, mapping from memory locations to symbolic values and their constraints) while the program location represents the exact symbolic execution location (e.g., before/after a statement) and the symbolic stack frames.

Symbolic execution of the program allows the analyzer to find deep, rare bugs that may be missed during the testing process. However, it is not a universal solution that finds all bugs; rather it provides a framework that can be engineered to find particular bugs. Specialized tools created by adapting the framework to seek specific kinds of bugs are called 'Checkers'. While the core analyzer framework executes the program in a symbolic manner, the checkers subscribe to different events (via callbacks to different program points), check various assumptions on symbolic values at these events, and throw errors/warnings if those assumptions fail on a given program path. Checks that are based on symbolic execution along a program path are called 'path-sensitive checks'. We used this framework to develop a custom checker to detect specific OpenSHMEM bugs.

The clang static analyzer provides the ability to write another type of check that exploits syntactic information only. Called 'AST-based checks', they are easy to write and add very little overhead to the compilation process. However, since they primarily use information available in Clang's Abstract Syntax Tree (AST) their ability to find bugs is limited to finding illegal or undesirable code patterns. The OpenSHMEM bugs we are trying to detect cannot be identified by an inspection of the AST alone and hence we chose to base our checker on the Clang Static Analyzer and its symbolic execution engine.

IV. Framework

We designed the OpenSHMEM Checker in such a way that it is easy to adapt for other PGAS programming models with minimal effort. It is comprised of 3 main components.

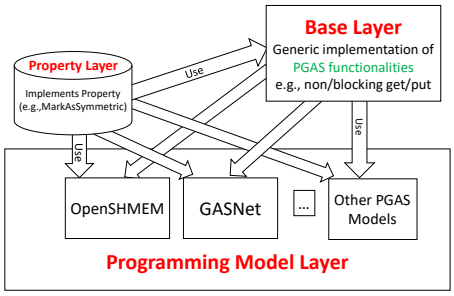

Fig. 1: OpenSHMEM Checker framework

A. Property Layer

OpenSHMEM introduces several PGAS library-specific behaviors/properties (e.g., whether a variable is symmetric or private) to the program data structures. These properties are often prerequisites and necessary for different OpenSHMEM routines to work (e.g., OpenSHMEM communication routines can only access the symmetric memories of a remote process, trying to access private memory through these routines would result in undefined behavior). OpenSHMEM routines also introduce different property-induced states to program data structures (e.g., synchronized and unsynchronized state of symmetric variables). These properties, associated states, and their transitions dictate an OpenSHMEM program behavior; therefore, modeling these properties, states, and their transition correctly is vital for finding programming errors.

Let us revisit the example in Listing 3 to explain how these properties and their states transition through a program and dictate the program behavior. For simplicity, we will focus on only one property, 'symmetric memory', and its states (synchronized and unsynchronized). In line 3, we use the OpenSHMEM allocation routine `shmem_malloc` to allocate memory in the heap. This is no ordinary heap-allocated memory (e.g., using C `malloc`); this memory has one extra property; it is symmetric and can be accessed by other processes directly. So, `shmem_malloc` introduces the 'symmetric memory' property to the allocated variable. Declaring variables as 'Static' or 'Global' in an OpenSHMEM program also introduces the 'symmetric memory' property to those variables.

Symmetric memories have different types of states associated with them (e.g., synchronization states, allocation states). Each of these types can have multiple state-values (e.g., synchronization type has two states, synchronized and unsynchronized). These states are used to model the OpenSHMEM programming model behavior, and transition to and from these states is dictated by OpenSHMEM routines.

TABLE II: Comparison of programming mistakes in MPI and OpenSHMEM

Programming Error	MPI	OpenSHMEM
Type mismatch	Exists (Data type vs. MPI type)	N/A - OpenSHMEM is type aware
Incorrect buffer referencing (e.g., referencing a 'int' buffer as character)	Exists	N/A - OpenSHMEM is type aware (few exceptions exist)
Invalid use of programming model specific objects	Invalid communicator, groups, and operations	Invalid context, and team
Invalid use of different types of memory	N/A, MPI does not introduce any new memory concept	Exists, wrong use of symmetric vs. private memory
Unmatched point to point call	Exists	N/A - OpenSHMEM uses one-sided communication semantics
Deadlock due to missing synchronization	Exists	Exists
Potential performance degradation due to excessive synchronization	Exists	Exists
Double non-blocking Unmatched wait Missing wait	Exists	Similar errors can happen in OpenSHMEM but in a different way. They fall in the category of 'Unsafe/unsynchronized access to program data'

Let us use Listing 3 again to see how these states transition in an OpenSHMEM program. In line 7, `shmem_put` is used to write data 'src' to PE 1's symmetric memory 'symMem'. For this routine to work correctly, it must fulfill the pre-condition that the first argument must be a symmetric memory. Once the pre-condition is fulfilled, and the routine is executed, it changes the synchronization state of the destination variable 'symMem', it makes 'symMem' at PE 1 'unsynchronized'. The reason for 'symMem' becoming 'unsynchronized' or unsafe is down to the communication nature of `shmem_put`. Since `shmem_put` returns right after the data has been copied out of the source array on the local PE, the delivery of the data is not guaranteed upon return from this routine. As a result, 'symMem' in PE 1 can have the old, new, or combinations of these values, rendering it to a non-deterministic or 'unsynchronized' state. Therefore without further synchronization routines to guarantee the delivery of the data, 'symMem' would be 'unsynchronized' or unsafe for further access (possible race condition). This transition of 'symMem' to 'unsynchronized' state is the result of `shmem_put`. Consequently, a global synchronization (`shmem_barrier_all`) or a local synchronization (`shmem_quiet`) routine at PE 1 guarantees that the new data has been written to 'symMem' and thus transitions to synchronized.

We implement these generic PGAS library-specific behaviors/properties and their associated states using the Clang Static Analyzer interface. We utilize CSA to do the bookkeeping for different properties and states associated with a certain data structure at different program points. We expose simple interfaces to add, remove, and query different properties and their states that can be used in the pre- and post-condition callbacks of the checker to detect errors. This property layer acts as an interface between the compiler and the library. Some of the interfaces implemented in the property layer are: `MarkAsSymmetric`, `IsMemRegionSymmetric`, `IsSynchronized`, `MarkAsSynchronized`, `MarkAsUnSynchronized`, `IsFree`, `IsArgNonNegative`.

B. Base Layer

We use the properties from the Property Layer to implement checks for generic PGAS operations such as blocking and non-blocking PUTs/GETs, memory allocation, and barriers. A specific PGAS library (e.g., OpenSHMEM) can use these checks out of the box, or provide its own implementations. Routines in the API can have pre-checks, post-checks, or both. In the pre-check phase, one can check for properties that must be fulfilled for the PGAS routine to work correctly (e.g., the destination variable of PUT must be a symmetric variable). In the post-check phase, one can check for a specific property after the PGAS routine is executed or specify actions that have to happen (e.g., after a PUT the destination variable becomes unsynchronized). So, a generic implementation of PUT would look like this:

```
API: PUT (dest, src, PE, ...);
Pre-checks: IsSymmetric(dest); .....;
Post-checks: MarkAsUnSynchronized(dest);
```

The primary purpose of the Base Layer is to make the analyzer extensible for other PGAS programming models. However, in this work we only focused on OpenSHMEM, other programming models were not considered.

C. Programming Model Layer

We implement programming model-specific operations in this layer. For example, in OpenSHMEM, we implement checks for routines like `shmem_put`, `shmem_get`, `shmem_malloc`, `shmem_barrier_all`. We can either use the default implementation provided by the Base layer for that specific type of routine, or we can provide our own implementation using the property layer APIs. A implementation of `shmem_put` would look like this:

```
API: shmem_put(T *dest, const T *source,
               size_t nelems, int pe);
Pre-checks: IsSymmetric(dest);
            IsArgNonNegative(pe);
Post-checks:MarkAsUnSynchronized(dest);
```

D. Why Layered Design

Compiler development has a steep learning curve, which often deters library developers from writing checkers for their libraries. This layer-based design of the OpenSHMEM Checker tries to alleviate that problem by allowing the developers to write a checker for a specific PGAS programming

model with minimal effort. It minimizes the interaction of the checker developer with the compiler, and allows the developer to use the default implementation out of the box from the base layer, or use properties implemented in the property layer to write their own library-specific checker. The layered design also allows modeling (adding support) of specific APIs or a group of APIs separately and independently; this enables development of incremental support for available APIs in a programming model without compromising the checkers ability to detect bugs in already supported APIs.

V. Integrated Checks

In this work, we implemented checks to detect the first 3 types of programming errors described in Section II-B. These checks are supported on a large number of frequently used OpenSHMEM routines. Currently supported routines cover most of the memory management, remote memory access, collective operations in OpenSHMEM. In this section, we briefly describe how these checks are implemented.

A. Violation of OpenSHMEM Semantics- Using Wrong Kind of Memory

To implement this check, we track all the variables with the symmetric memory property (global and static variables, and memory managed by the OpenSHMEM allocation routines). Heap-allocated variables are marked using the property layer interface `MarkAsSymmetric` in the 'POST_CALL' callbacks of OpenSHMEM allocation routines. They are marked immediately after they are allocated, irrespective of their future use. It is not easy to mark global and static variables from their declaration, so we wait until their first use in an OpenSHMEM routine and mark them as symmetric using the `MarkAsSymmetric` routine. We track these symmetric variables using a path-sensitive `ImmutableMap`.

In the 'PRE_CALL' callback of each OpenSHMEM routine that takes symmetric arguments, we check that symmetric property using the property layer interface `IsSymmetric`.

`IsSymmetric` searches the `ImmutableMap` that contains the list of all symmetric memories, and determines whether the variable is symmetric. If not, we raise a bug and generate a report for 'wrong kind of memory.'

During the tracking process, we also track if a symmetric memory is heap-allocated or not, as some routines require this (e.g., `shmem_realloc`).

B. Double-free

To implement double-free of the symmetric variables allocated via OpenSHMEM, we track the allocation and de-allocation of each symmetric variable in all program paths. Once they are allocated, we change their state to 'allocated' by using the property layer interface `RecordThisAllocation` in the 'POST_CALL' callback of the allocation routine. Once a variable is freed using `shmem_free`, we change the state of that variable to 'freed' using `FreeThisAllocation` from the property layer. As part of the process, `FreeThisAllocation` checks if it has been freed; if so, it raises a 'double-free' error and reports a bug.

C. Unsynchronized Access to Program Data

This check provides experimental support for detecting unsynchronized access to program data. In its current form, it works best if the memory region that is affected by the unsynchronized access is constant-reducible (e.g., after constant propagation, it results in a constant value). This is due to: 1) the inherent limitation of static analysis (some information is only available at runtime), 2) current status of region arithmetic in CSA which is still being improved, and 3) the lack of parallel communication analysis (notion of parallelism) in CSA. Improving the later two would enable our checker to support non-constant memory regions.

To implement this check, we used 'ranges' (start to end) to track unsynchronized memory regions. Every time a portion of the memory becomes unsynchronized due to an OpenSHMEM call (e.g., `shmem_put`), we use symbolic expressions (LLVM `SVal` objects) to store the start-index and number of elements read/written, from which we derive the start-index and the end-index of that portion of the memory using the `SValBuilder`. We use this approach because most OpenSHMEM routines use an 'offset' and 'numElements' to read from and write to a symmetric memory. We also use this information to change a memory region's status from unsynchronized/unsafe to synchronized.

Beyond this, we also track the PE in which the unsynchronized array region resides in, thus providing exact information as to which memory is unsynchronized and removing false-positives in the process.

VI. Evaluation

We used a custom test suite consisting of synthetic benchmarks dedicated for each error type and 3 Benchmark applications to evaluate the OpenSHMEM checkers performance. The benchmark applications, *Transpose* transposes a $N \times N$ matrix using a blocked approach, *Matrix Multiplication (MM)* multiplies two matrices based on Cannon's algorithm, and *Mandelbrot Set* generates a greyscale image of the Mandelbrot set using the quadratic iteration function. We introduced different programming bugs into the working versions of these applications. Due to the lack of available benchmark applications in OpenSHMEM, we developed two (Transpose, and MM) of the ones used here.

We evaluated the checker's ability to find the 3 types of bugs it supports and the overhead. We used LLVM version 10.0 in a machine with Intel(R) Xeon(R) Gold 5115 CPU @ 2.40GHz with 192GB of memory running Fedora 32.

A. Case Study: Synthetic Benchmarks

We developed synthetic benchmarks that utilize different OpenSHMEM routines supported by the checker and use it to test different programming errors.

1) Wrong Kind of Memory: We introduced 'wrong kind of memory' bugs in various OpenSHMEM routines, and the checker was able to detect all of them. Since the checker utilizes aliasing information, it was able to detect bugs that involved pointer following, which is illustrated in Listing 4.

```
1  int *GlobalPtr;     //Global (symmetric) variable
2  int main(int argc, char *argv[])
3  {
4      ...
5      int *privMem = (int *) malloc(N*sizeof(int));
6      int *source = (int *) malloc(N*sizeof(int));
7      // GlobalPtr is a symmetric variable
8      // but it points to a private memory
9      GlobalPtr = privMem;
10     // Error: GlobalPtr is expected to be symmetric
11     shmem_put(GlobalPtr,source,1,pe);
12     ...
13 }
```

Listing 4: Wrong Kind of Memory Error

Here, `GlobalPtr` is a symmetric variable (line 1), however, it points to private memory (line 9) allocated by `malloc`. When it is used as a destination variable (first argument of `shmem_put` which OpenSHMEM expects to be symmetric memory), the checker catches and throws the appropriate error.

2) Double-free: We implemented double-free errors in several different ways and the checker was able to detect them successfully. These included the use of pointer indirection (aliasing) as well as an inter-procedural Double-free error. One scenario is illustrated in Listing 5.

```
1      ...
2      int *ArgPtr =(int *)shmem_malloc(N*sizeof(int));
3      ...
4      int *indirectArgPtr = ArgPtr;
5      ...
6      shmem_free(indirectArgPtr);     //Free 1
7      ...
8      // Error: Double-free, indirectArgPtr and ArgPtr
9      // points to the same memory which has already
10     // been freed previously by freeing
11     // indirectArgPtr
12     shmem_free(ArgPtr);             //Free 2
13     ...
```

Listing 5: Double-free Error

3) Unsynchronized Access: The checker was not able to detect all of the unsynchronized access errors that we introduced into the synthetic benchmark. In order to avoid false-positives, we currently only detect unsynchronized accesses whose indexes and the PE numbers can be propagated to constants. Where these conditions do not hold, the checker fails. Comprehensive communication and dependence analyses would improve the detection of this error. Listing 6 shows both detected and undetected errors.

B. Case Study: Transpose, MM, Mandelbrot

We also evaluated the checkers ability to find Wrong kind of memory, Double-free, and Unsynchronized access errors that were injected into the Transpose, MM, and Mandelbrot set benchmarks. Before introducing the bugs in these programs, we tested them 'out of the box' to check for false positives, however, the tool did not find any false positive during this process. Due to the lack of space, we only provide the summary of the results (injected and caught bugs). However, we followed the approaches described for the synthetic benchmarks to introduce bugs in these programs which should enable the readers to re-produce results obtained here.

TABLE III: Overhead analysis. Compile-time shown in secs.

Benchmarks	No Static Analysis (s)	With Static Analysis (s)	With OpenSHMEM Checker (s)
Transpose	5.69	5.81	5.83
MM	4.78	5.43	5.50
Mandelbrot	0.68	28.95	33.91

1) Wrong Kind of Memory: The checker was able to detect 4 out of 5 wrong kind of memory errors. It missed one error in 'Transpose' simply because the OpenSHMEM routine in question, `shmem_fcollect` is not currently modeled (supported) by the Checker. That did not affect the ability of the checker to detect other errors due to its layered design. We plan to add support for all OpenSHMEM routines in the future.

```
1  int main(int argc, char *argv[])
2  {
3      ...
4      int *symMem = (int *) shmem_malloc(N*sizeof(int)
       );
5      int pe = shmem_my_pe();
6      if(pe != 0) {
7          shmem_put(&symMem[0], source, 1, 0);
8          ...
9          // Error: Unsynchronized Access to symMem
10         shmem_get(source, &symMem[0], 1, 0);
11     }
12     shmem_put(&symMem[pe], source, 1, pe);
13     ...
14     // Unsynchronized Error not detected since value
15     // of 'pe' is not a constant
16     shmem_get(source, &symMem[pe], 1, pe);
17     ...
18 }
```

Listing 6: Unsynchronized access Error

2) Double-free: The checker detected all 4 injected Double-free errors. One Double-free scenario in Transpose was inter-procedural and the checker handled it successfully.

3) Unsynchronized Access: We introduced an Unsynchronized access error into MM. The checker was unable to detect it because the array indexes and PE numbers used in the program were not propagated to constants. Since this shortcoming can hamper the checker's ability to detect Unsynchronized access errors in real world applications, we will work on the necessary support analyses in future.

C. Overhead Analysis

OpenSHMEM Checker is a static analysis tool; hence it does not have any runtime overhead. Our checker adds a small to negligible compilation overhead on top of the Clang Static Analyzer Core if any other path-sensitive checker is used. The overhead is mostly related to the generation of the exploded graph and the number of paths that need to be explored. Therefore, the overhead is dependent on the number of conditionals in a program rather than the code size. Table III shows the overhead result for the 3 benchmark applications. We compare compilation time without static analysis, with static analysis (with the default checkers), and finally with the OpenSHMEM Checker enabled with other default checkers. Among the benchmarks, Mandelbrot has the most conditional statements, resulting in a large exploded graph and a significant increase in compile-time if the static analysis is used with a small overhead added on top of that using the OpensHMEM Checker.

VII. Related Work

This work has roots in three overlapping research areas:

A. Program Correctness of Parallel Programs

A variety of methods have been explored to detect and remove bugs from parallel applications, including runtime error-verification, trace-based error detection, model checking and static analysis. Each has its strengths and weaknesses.

Runtime approaches usually utilize instrumented code to check for anomalies in the runtime behavior. Such systems include UPC-Check [10], MPI-Check [11], MARMOT [1], [12], UMPIRE [2]. While these techniques are specially good for detecting runtime bugs such as race conditions, deadlock, erroneous arguments, they add over-head in the runtime. Also they focus on a specific part of the program and are agnostic regarding the program control structure. As a result, finding the source of the error may be difficult, since the position where the error is detected and the source of the error may not be the same. In contrast, our work uses static analysis techniques.

Trace-based error detection tools such as BoundsChecker [13], the Intel Message Checker [14] analyzes trace files to detect errors such as mis-matched buffer types, race conditions and deadlocks. Trace-based techniques work well with 2-sided communication as in MPI (matching send-recv pair), but detecting errors in 1-sided communication models such OpenSHMEM can be extremely difficult.

Another error detection technique is model checking, which uses formal methods to check the validity of a program. The user typically models the input/output, logically represents the program as a finite state model, and makes assertions for different states using a modeling language: MPI-SPIN [15], UPC-SPIN [16], MAGIC [17], SLAM [18] use this technique.

B. Static Analysis Techniques for Program Correctness

Static analysis is a popular method to check for program correctness. It uses static program information (e.g., compile-time information) and symbolic execution to find errors in a program. It usually exploits existing compiler information. Droste et al. [3], Ye et al. [19], Yu et al. [20] use LLVM's Static Analyzer while Aananthakrishnan et al. [21] use ROSE compiler framework. However, these mostly focus on MPI.

C. OpenSHMEM Program Analysis

Work in the area of program analysis and error detection for OpenSHMEM is scarce. The OpenSHMEM Analyzer [22], [23] is the only prior work in this domain. Like our work, it uses compiler-based static analysis to detect bugs in OpenSHMEM applications. However, the OpenSHMEM Analyzer is built on top of OpenUH (a branch of the Open64 compiler) which is no longer supported. Our checker is based on the popular LLVM framework, which has a modular infrastructure that makes our work easily extensible.

VIII. Conclusion and Future Work

We have developed a CSA-based static checker to help find bugs in OpenSHMEM programs. The checker has a layered design to make it easy to use, extend, and modify. We also made sure that no false-positives are reported by the checker, so that programmers can use the report with confidence. Early evaluation result shows great promise. However, much needs to be done to provide a comprehensive error detection mechanism for OpenSHMEM and other PGAS programming models.

We are extending our work to enable unsynchronized access error detection, which requires comprehensive communication and dependence analysis. We also plan to add support for other errors such as over synchronization, and deadlocks and ultimately to support the entire OpenSHMEM specification.

Acknowledgment

This material is based upon work supported by the National Science Foundation under grant no. CCF-1725499. The authors also thank Stony Brook University for access to the HPC systems.

References

[1] B. Krammer *et al.*, "Marmot: An mpi analysis and checking tool," in *Advances in Parallel Computing*. Elsevier, 2004, vol. 13, pp. 493–500.
[2] J. S. Vetter and B. R. De Supinski, "Dynamic software testing of mpi applications with umpire," in *SC'00*.
[3] A. Droste *et al.*, "Mpi-checker: static analysis for mpi," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 2015.
[4] B. Chapman *et al.*, "Introducing openshmem: Shmem for the pgas community," in *PGAS 2010*.
[5] O. Community, "Openshmem application programming interface version 1.5," http://www.openshmem.org/, 2020.
[6] T. S. Woodall *et al.*, "High performance rdma protocols in hpc," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, 2006.
[7] G. F. Pfister, "An introduction to the infiniband architecture," *High performance mass storage and parallel I/O*, vol. 42, no. 617-632, p. 102, 2001.
[8] J. Jose *et al.*, "Designing scalable graph500 benchmark with hybrid mpi+ openshmem programming models," in *ISC 2013*.
[9] Z. Xu *et al.*, "A memory model for static analysis of c programs," in *ISoLA 2010*.
[10] J. Coyle *et al.*, "Upc-check: a scalable tool for detecting run-time errors in unified parallel c," *Computer Science-Research and Development*, vol. 28, no. 2-3, pp. 203–209, 2013.
[11] G. Luecke *et al.*, "Mpi-check: a tool for checking fortran 90 mpi programs," *Concurrency and Computation: Practice and Experience*, vol. 15, no. 2, pp. 93–100, 2003.
[12] B. Krammer and M. M. Resch, "Correctness checking of mpi one-sided communication using marmot," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, 2006.
[13] M. Focus, "Boundschecker," 2021.
[14] J. DeSouza *et al.*, "Automated, scalable debugging of mpi programs with intel® message checker," in *SE-HPCS '05*.
[15] S. F. Siegel, "Verifying parallel programs with mpi-spin," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 2007, pp. 13–14.
[16] A. Ebnenasir, "Upc-spin: A framework for the model checking of upc programs," in *PGAS11*.
[17] S. Chaki *et al.*, "Efficient verification of sequential and concurrent c programs," *Formal Methods in System Design*, vol. 25, no. 2, pp. 129–166, 2004.
[18] T. Ball *et al.*, "Slam and static driver verifier: Technology transfer of formal methods inside microsoft," in *IFM 2004*.
[19] F. Ye *et al.*, "Detecting mpi usage anomalies via partial program symbolic execution," in *SC' 18*.
[20] H. Yu, "Combining symbolic execution and model checking to verify mpi programs," in *ICSE '18*.
[21] S. Aananthakrishnan *et al.*, "Parfuse: Parallel and compositional analysis of message passing programs," in *LCPC 2016*.
[22] O. Hernandez *et al.*, "The openshmem analyzer," in *PGAS12*, 2012.
[23] S. Pophale *et al.*, "Extending the openshmem analyzer to perform synchronization and multi-valued analysis," in *OpenSHMEM 2014*.