Checkpointing OpenSHMEM Programs Using Compiler Analysis

Md Abdullah Shahneous Bari*, Debasmita Basu*, Wenbin Lu*, Tony Curtis*, and Barbara Chapman*†
{MdAbdullah.ShahneousBari, Debasmita.Basu, Wenbin.Lu, Anthony.Curtis, Barbara.Chapman}@stonybrook.edu
*Stony Brook University, †Brookhaven National Laboratory

***Abstract*—The importance of fault tolerance continues to increase for HPC applications. The continued growth in size and complexity of HPC systems, and of the applications themselves, is leading to an increased likelihood of failures during execution. However, most HPC programming models do not have a built-in fault tolerance mechanism. Instead, application developers usually rely on external support such as application-level checkpoint-restart (C/R) libraries to make their codes fault tolerant. However, this increases the burden on the application developer, who must use the libraries carefully to ensure correct behavior and to minimize the overheads. The C/R routines will be employed to save the values of all needed program variables at the places in the code where they are invoked. It is important for correctness that the program data is in a consistent state at these places. It is non-trivial to determine such points in OpenSHMEM, which relies upon single-sided communications to provide high performance. The amount of data to be collected, and the frequency with which this is performed, must also be carefully tuned, as the overheads introduced by C/R calls can be extremely high.**

There is very little prior work on checkpoint-restart support in the context of the OpenSHMEM programming interface. In this paper, we introduce OpenSHMEM and describe the challenges it poses for checkpointing. We identify the safest places for inserting C/R calls in an OpenSHMEM program and describe a straightforward approach for identifying the data that needs to be checkpointed at these positions in the code. We provide these two functionalities in a tool that exploits compiler analyses to propose checkpoints and the sets of data for saving at them, to the application developer.

***Index Terms*—Fault Tolerance, Check-pointing, Compiler Assisted Check-pointing, Compiler Analysis**

I. Introduction

Determining how to build and deploy computer systems to enable recovery from failure, and what importance to place upon such capabilities (e.g., architectural redundancy in the network), is a perennial problem for computer systems designers. The greater the size and complexity of a system, the more likely it is that one or more components will fail during the execution of an application code. HPC (High Performance Computing) systems are inherently large and with the growing heterogeneity of processors and memory subsystems they are also increasingly complex. In the race to higher computational power, the number of components continues to increase and in consequence, the Mean Time Between Failure (MTBF) of these systems is decreasing [1], [2]. The smaller the MTBF, the less stable the system is. Especially for long-running applications, whose execution may take significantly longer than the MTBF of the platforms they are deployed on, it is vital that the interim results of a computation are not lost if a system fault occurs. This could otherwise lead to not just wasted computing resources, but potentially greater problems if the results are time-sensitive (e.g., weather forecasting). Thus strategies for resilience - in hardware, system software, and application software - are essential in HPC. Since an application program may run on a variety of platforms with differing MTBFs, application developers who need to ensure that failure does not compromise the timely production of results must pro-actively adopt a strategy to accomplish this. In other words, in most HPC contexts, the safe approach is to build resilience into the application code.

Unfortunately, most traditional HPC programming models (e.g., MPI, OpenSHMEM, GASNet) lack the support for resilience that is built into their Data Analytics counterparts (e.g., Hadoop, Spark). Application developers must therefore rely on other means to achieve fault tolerance. Established techniques include but are not limited to Checkpoint-Restart (C/R), containment domains, and replication of computations. These techniques can be employed at the application level, at the system level or both. Among them, Checkpoint-Restart (C/R) is the most-used approach to implement fault tolerance in HPC applications due to its versatility, and relatively low overhead.

Checkpoint-Restart follows a simple strategy; one saves the state of the program as a so-called checkpoint at certain intervals during the execution. If the program terminates for any reason (hardware or software failure/faults) prior to completion, it restarts from the last checkpoint using the saved information instead of starting again from the beginning. Since the successful restart of an application relies on the data output during the check-pointing process, saving enough information to restart is very important.

However, care must be taken with respect to the amount of data saved during the check-pointing process. Since check-pointing results in a lot of data movement, oftentimes across nodes, the communication and I/O overhead it incurs can potentially result in severe performance degradation. Hence it is very important, but also extremely difficult, to determine the right amount of information to save, and a reasonable frequency at which to do so.

Moreover, there is another important challenge involved in implementing the Checkpoint-Restart process, that of *finding suitable places to checkpoint*. One has to make sure that the program is in a consistent and deterministic state with respect

to both computation and data when the state of the program is saved at a checkpoint, otherwise it may result in a non-deterministic and potentially incorrect restart of the program. Finding these consistent and deterministic "safe points" can be tricky in an application that uses a parallel programming model such as MPI. However, it can be significantly tougher for an application that uses a programming model with asynchronous communications such as OpenSHMEM and other PGAS programming models (e.g., GASNet, UPC++).

Hence, the successful deployment of C/R depends on solving these consistency and data optimization challenges efficiently and effectively. However, to do so one has to have a good understanding of not just the programming model, and the checkpointing library being used, but also of the entire application, which can be very difficult for large scale scientific applications that have hundreds of thousands of lines of code. Earlier research has coupled an understanding of the semantics of a programming interface with compiler techniques [3]–[9] to ease the burden on application developers by identifying safe points for checkpointing, suggesting which data to save and so on. However, prior work of this kind has primarily focused on providing support for MPI and not for other programming models such as OpenSHMEM. OpenSHMEM's user base is growing, especially as a result of its benefits for computations with an irregular communication pattern (e.g., graph based applications). This is a result of its reliance on asynchronous, one-sided communications along with RMA (Remote Memory Access) support in recent hardware, and its careful implementation. Hence, support for developing resilient OpenSHMEM applications is needed. Yet the features that provide some of its key benefits lead to challenges when approaches created for MPI are applied to it.

In this paper:

- We analyze the OpenSHMEM programming model from a resilience perspective and define "safe points" for inserting checkpointing library calls into OpenSHMEM applications.
- We use a compiler's data analysis to determine what data to checkpoint.
- We describe a tool that we have created to support the insertion of checkpoints into OpenSHMEM code. It utilizes a safe point analysis and data analysis to provide check-pointing suggestions (where to checkpoint, what data to checkpoint) to the OpenSHMEM application developer. The tool is based on an open-source compiler framework, LLVM [10], which is becoming very popular in both academia and industry.

II. Background

A. PGAS Programming Models

The Partitioned Global Address Space (PGAS) [11] is a parallel distributed programming model that typically 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. PGAS models often take advantage of network capabilities such as Remote Direct Memory Access (RDMA) [12] to allow efficient data movement that is decoupled from synchronization. The PGAS family includes libraries and languages: the former include Global Arrays, GASNet, and OpenSHMEM; and the latter, which are built on these libraries, include UPC, UPC++, and Fortran's Co-Arrays.

B. OpenSHMEM

The OpenSHMEM[1] Specification [13], [14] defines a library-based PGAS programming model/interface for C and C++. 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 [15], Ohio State University [16], Sandia National Lab [17], Oak Ridge National Lab [18], Open-MPI [19]; and some from vendors, e.g. HPE/Cray [20], NVIDIA/Mellanox [21], IBM [22].

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 operations; and collective memory management, communication, and synchronization operations.

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 [23], 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. Figure 1 shows the OpenSHMEM memory model.

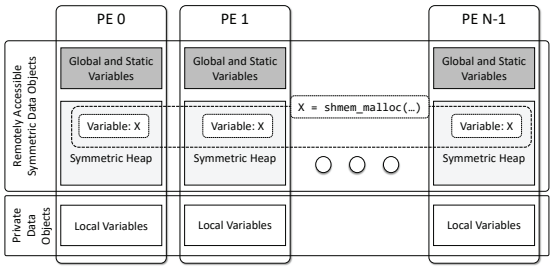

Fig. 1. Memory model in OpenSHMEM. The remotely accessible data consists of: 1) Global or static variables 2) Data on the symmetric heap (allocated by shmem_malloc)

2) Remote Memory Access Routines: Two of the main OpenSHMEM RDMA routines are the generic forms `shmem_put` and `shmem_get`[2], which allow 1 PE to respec-

[1]The OpenSHMEM copyright is owned by Open Source Software Solutions Inc., a non-profit organization, under an agreement with Hewlett Packard Enterprise.

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

tively 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 [24]. 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, this introduces challenges with respect to providing fault tolerance support to OpenSHMEM applications. One of those challenges is finding points (safe points) in a program where the memories are in a consistent and deterministic state.

C. Checkpoint-Restart Methods for Fault Tolerance

Checkpoint-based rollback recovery methods can be primarily categorized into three main groups: uncoordinated checkpointing, coordinated checkpointing, and communication-induced checkpointing [25].

In ***uncoordinated checkpointing***, each process is in charge of its own checkpoints and is allowed to take them at its own convenience without coordinating with other processes. As a result, coordination overhead during the checkpointing process is reduced. It also reduces synchronization cost and energy consumption during the recovery process [26] since a single process crash does not result in all processes reverting back to the last checkpoint and recomputing. However, since there is no coordination, each process might have to keep multiple checkpoints which could be detrimental to performance. These protocols are also subject to domino effects and do not guarantee progress [25].

In **Coordinated checkpointing**, where all processes coordinate their checkpoints to form a consistent global state, is free of domino effects and has a simple recovery process. However, in the case of a large number of processes, these protocols may suffer from scalability issues due to the necessity of global coordination. Researchers have explored different approaches such as, non-blocking checkpoint coordination, checkpoint with synchronized clocks, minimal checkpoint coordination, to alleviate this problem.

Communication-induced checkpointing (CIC) protocols are a bridge between uncoordinated and coordinated C/R protocols. These protocols avoid the domino effect without requiring all checkpoints to be coordinated. In CIC, processes take two kinds of checkpoints: local and forced. Local checkpoints can be taken independently as with uncoordinated protocols, while forced checkpoints must be taken to guarantee progress. Forced checkpoints are not coordinated checkpoints; rather, they are taken at the discretion of each individual process based on the communication pattern and checkpoint information received from other processes. However, these protocols are unpredictable due to their dependence on the communication pattern of the application [27].

Among these protocols, coordinated checkpointing techniques are the simplest and often work best for PGAS programming models [25], [28].

All these protocols can be implemented either at the system level or at the application level. They differ in the abstraction level at which the state of a process is saved. In system-level checkpointing, the entire state of the process, such as the contents of the program counter, registers, and memory are saved [29]. However, the amount of data saved during system-level checkpointing can be extremely large and the overhead may impact the application performance adversely. As a result, system-level checkpointing is not a popular choice for large-scale HPC platforms [5].

In application-level checkpointing, applications implement their own checkpointing code, usually by means of a third party checkpointing library [30]–[32]. We assume the use of such a library in the following. Applications decide which data to checkpoint and at what points in the program to do so. As a result, the amount of data to be checkpointed can be significantly reduced, since the application developer can decide exactly how much data needs to be saved, and moreover, system-level data is not saved. Application-level checkpointing also provides the added cushion of portability (not tied to a specific system). As a result, it is a popular choice for large-scale HPC systems. However, it places a huge burden on the developer to choose the right data and right place to checkpoint, including ensuring that the program state is consistent at such locations.

D. Application-level Checkpointing and Challenges

Although popular, application-level checkpointing comes with its own specific challenges, of which two important ones are maintaining program correctness, and controlling checkpointing overhead.

1) Program Correctness: Program correctness must not be compromised when using an application-level checkpointing

scheme. To maintain the correctness of the program, checkpointing calls must be inserted into the program at places in the code where the program and its data are in a deterministic and consistent state. Finding these points in an application can be non-trivial and the complexity may vary across programming models since it is dependent on the semantics of features of the language or library. Section III discusses this issue further and outlines how it can be handled for OpenSHMEM.

2) Checkpointing Overhead: Checkpointing overhead is a major issue for any checkpointing scheme. Since checkpointing involves saving or copying a large amount of data (program state) to a persistent storage or memory, the overhead associated with this process can be extremely large. To reduce this overhead different optimization techniques are often used.

Three major optimizations utilized in practice are:

- **Minimizing the amount of data to be saved at a certain checkpoint:** In a C/R scheme, we only need to save the data that are necessary for restart, and, not all data are necessary at every point of the program. Hence, to minimize the amount of data to be saved at a certain checkpoint, different techniques that utilize compiler program analysis or operating system support are used in practice. Some of these techniques are, incremental checkpointing, region exclusion, and live variable analysis.
- **Choosing, and potentially adapting, the frequency of the checkpoints:** Having checkpoints too frequently may result in overhead explosion, while too few checkpoints may result in losing a lot of computation in case of a failure. Hence, choosing the optimal checkpointing interval is important. Checkpointing interval depends on both the structure of the application and the MTBF of the system it is running on. Finding optimal places for checkpointing is an active research area.
- **Optimizing the way the data is saved:** Optimizing the way the checkpointing data is saved in persistent storage or in memory may result in a significant overhead reduction. Several optimization techniques such as buffering, where intermediate storage such as burst buffer is used, and saving different data to different types of storage, exist in practice.

In this work, we focus primarily on the first optimization. Details of the optimization are described in Section IV.

III. OpenSHMEM and Safe Points

A. Global Consistency and Safe Points

When checkpointing an OpenSHMEM application (or any other parallel application for that matter), one has to make sure that the checkpoint is taken at a point where the application's computation, communication, and memory are in a consistent and deterministic state in order to ensure deterministic recovery. We call these points "safe points". Due to the relaxed memory consistency nature of OpenSHMEM's memory model, non-blocking accesses to a PE's symmetric memory between two synchronization points may result in an inconsistent state of the PE's symmetric memory.

We use the code example shown in Figure 2 to explain this in the context of an OpenSHMEM program. The code snippet shows a part of a halo exchange, where PEs share the updated values of elements at the edges of their sub-regions with neighboring PEs. For simplicity we show the interaction between two neighboring processes only. Here, PE0 (the host PE) uses `shmem_put` to write/copy the first two elements of its **host_halo** array to the first two elements of PE1's **neighbor_halo** array; **neighbor_halo** has been allocated in symmetric memory so that PE0 can access it directly via OpenSHMEM routines. However, between the call of `shmem_put` and that of `shmem_barrier_all`, the symmetric array **neighbor_halo** of PE1 is in an inconsistent state since there are 3 possible states of **neighbor_halo** (shown by 3 different scenarios in Figure 2). These states are no update (Scenario 1), partial update (Scenario 2) and complete update (Scenario 1). So if a checkpoint is taken at this place in the code, the checkpointed data (**neighbor_halo**) would be in an unknown state, resulting in non-deterministic behavior during recovery. However, OpenSHMEM's synchronization routines (e.g., `shmem_barrier_all`, `shmem_quiet`) do confirm the completion of communication routines (e.g., `shmem_put`) and in OpenSHMEM guarantee consistency of some or all communications (no in-flight messages). The `shmem_barrier_all` routine used in this example is a global synchronization routine and it provides global consistency guarantees across all PEs. Hence, the point in the code immediately after `shmem_barrier_all` is a safe-point and a checkpoint taken there would guarantee deterministic recovery.

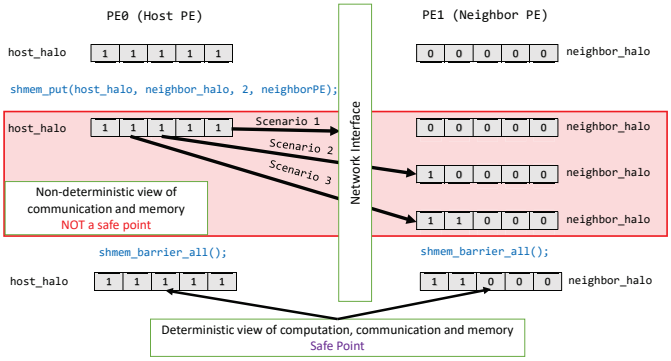

Fig. 2. An example showing a safe point in an OpenSHMEM program

B. Why Global Synchronizations are the Best Safe Points for OpenSHMEM

OpenSHMEM provides different types of synchronization or memory consistency routines: local consistency routines that only affect the operations initiated by the calling PE (`shmem_quiet`), and global consistency routines that affect all operations initiated by all PEs (`shmem_barrier_all`). In an ideal scenario, it is possible to reason about the local consistency routines across PEs and find safe-points. In a message-passing model such as MPI,

TABLE I
SAFE POINTS IN OPENSHMEM API. IT'S SAFE TO CHECKPOINT RIGHT AFTER A SUCCESSFUL CALL TO THESE FUNCTIONS.

Name	Description	Pre-requisite for Global Sync	Suggestion
`shmem_barrier_all`	Global barrier	None	None

`shmem_malloc`
`shmem_malloc_with_hints`
`shmem_calloc`
`shmem_align`
	Symmetric memory allocation collective operations that require participation by all PEs in the world team. Global synchronization on exit if the calls are successful.	Call to these functions have to be successful	Add a check to make sure the allocation is successful

`shmem_realloc`	Symmetric memory reallocation routine (collective). May use global synchronization on both entry and exit, depending on whether an existing allocation is modified and whether new memory is allocated, respectively.	call this function has to be successful	Add a check to make sure the allocation is successful
`shmem_free`	Symmetric memory de-allocation routine (collective). Global synchronization at the entry.	None	None

`shmem_init`
`shmem_init_thread`
`shmem_finalize`
	OpensHMEM library initialization and finalization routines (collective).	None	None

`shmem_quiet` /
`shmem_ctx_quiet`
+
`shmem_sync_all`/
`shmem_team_sync(world)`
	Combination of `shmem_quiet/shmem_ctx_quiet` routine called by all PEs, followed by a `shmem_sync_all` collective or `shmem_team_sync` (on the world team) results in global synchronization.	None	None

each process knows what data it is sending and what it is receiving (send-receive rendezvous pair). Hence, by performing a communication analysis for a certain code-block, it is possible to reason about whether the process's communication and data are in a consistent state or not. In contrast, PGAS models like OpenSHMEM are not based on send-receive communication pairs. An active PE in OpenSHMEM initiates communication to write to or retrieve data from the symmetric memory of another PE. Since the latter is not involved in the communication, it may not be aware that this is taking place. Hence it can not locally reason about its own data consistency, since it does not know if any communication initiated by other processes is in flight which would leave the process's symmetric memory in an inconsistent state. It is similarly hard for a compiler or tool to reason about the consistency of an application program at any given time.

Hence, the easiest and safest way to identify places in the code where data is in a consistent state in OpenSHMEM is to exploit the global consistency achieved immediately upon completion of global synchronization, where the network is quiet (no in-flight messages).

Table I shows the OpenSHMEM routines that provide global synchronization and can be used as safe points. However, OpenSHMEM has the concept of user-defined communication context or communication channel. A communication context is a container for communication operations. Each context provides an environment where the operations performed on that context are ordered and completed independently of other operations performed by the application. Each OpenSHMEM program comes with a default communication channel called "default context".

The safe points described in Table I are for default context. Applications with user-defined contexts must additionally ensure that those contexts are quiet (all the update/communication to symmetric memory is completed) before using those routines as safe points. Calls to `shmem_ctx_quiet` by all PEs would ensure the completion of outstanding communication to symmetric objects (would quiet that specific context).

IV. CHOOSING THE RIGHT DATA TO CHECKPOINT

In the previous section, we discussed the importance of finding safe points in order to ensure that, in the presence of a fault, the data saved in a checkpoint will enable a correct restart. We also identified OpenSHMEM routines that can be used for that purpose.

While finding the safe points, and ensuring that we insert checkpointing calls only at those points, solves the correctness issue, we still need to optimize the amount of data to be saved during the checkpointing process in order to avoid unnecessary overhead. If insufficient attention is paid to reducing the amount of data saved during a checkpoint, the overheads may become insurmountable. In order to alleviate this problem, researchers have investigated memory exclusion techniques both at compiler level [33] and at runtime level [34]. In memory exclusion, regions of a process's memory are excluded from a checkpoint because they are either read-only, meaning their values have not changed since the previous checkpoint, or dead, meaning their values are not necessary for the successful completion of the program [35].

We apply and adapt the results of a standard compiler analysis called "Live Variable Analysis" to identify variables (scalars and arrays) that we do not need to checkpoint at a specific position in the program code. Live Variable Analysis is traditionally applied to scalar variables in order to optimize register usage, where it helps remove variables from registers when they are no longer needed. Live variable (or liveness) analysis is a data flow analysis that essentially determines whether the current value of a variable may be used in the future. A variable *x* is live at a program point/statement *s* if some computational path from *s* to the end of the program (or function) contains a use of *x* which is not preceded by a new definition. In other words, it is live if the current value of variable *x* may be used at a later point in the code. The set of variables $LIVE_{in}[s]$ that are live at a given statement *s* can be calculated using the following formula:

$$LIVE_{in}[s] = (LIVE_{out}[s] - DEF[s]) \bigcup USE[s] \quad (1)$$

where $LIVE_{out}[s]$ is the set of variables live after the statement s, $USE[s]$ represents the set of variables used by s, and $DEF[s]$ represents the set of variables defined by s.

If a variable is not live at a certain checkpoint, then the value that it currently has is not used in any subsequent code. In other words, if the variable occurs at a later point in the code, then any uses of the variable will be preceded by new assignments to it. Therefore, we do not have to save it at that specific checkpoint. This can potentially help a user (or tool) identify and delete some unnecessary saves during the checkpointing process. Figure 3 shows a simple OpenSHMEM code segment that utilizes live variable analysis information for checkpointing purposes. The program has 8 safe points, `shmem_init` at line 3, `shmem_malloc` at line 4 and 5, `shmem_barrier_all` at line 13 and 20, `shmem_free` at line 27 and 28, and finally `shmem_finalize` at line 29. Here, we only consider `shmem_barrier_all` for checkpointing. At line 13, both *A* and *B* are live, hence, we have to save both variables for a successful restart (shown in line 14). At line 20, we see that only *B* is live, therefore we only need to save *B* instead of both *A* and *B* (shown in line 21). At this checkpoint, live variable analysis helped us reduce the checkpoint data to half.

```
1. int main()
2. {
3.   shmem_init();        //safe point (SP)
4.   int *A = (int *)shmem_malloc(N * sizeof(int));//SP
5.   int *B = (int *)shmem_malloc(N * sizeof(int));//SP
6.   ...
7.
8.   for(int i = 0; i<N ; i++)
9.   {
10.      A[i] = init_A(i);
11.      B[i] = init_B(i);
12.  }
13.  shmem_barrier_all();  //safe point; LIVE = {A, B}
14.  CHECKPOINT(A, B);

15.  for(int i = 0; i<N ; i++)
16.  {
17.      … = A[i];         //uses A
18.      …
19.  }
20.  shmem_barrier_all();  //safe point; LIVE = {B}
21.  CHECKPOINT(B);        //A is not live anymore

22.  for(int i = 0; i<N ; i++)
23.  {
24.      … = B[i];         //uses B
25.      …
26.  }
27.  shmem_free(A);        //safe point
28.  shmem_free(B);        //safe point
29.  shmem_finalize();     //safe point
30.  return 0;
31. }
```

Fig. 3. A skeleton OpenSHMEM program showing the impact of Live variable analysis for checkpointing

V. Compiler Tool: Putting it Together

We developed a tool that incorporates the OpenSHMEM-specific safe point analysis and data optimization discussed in Section III and IV respectively. The purpose of this tool is to provide a starting point for application-level checkpointing. It identifies the safe points in an OpenSHMEM program and provides suggestions as to what data to checkpoint at each of them in case the user decides to use a given safe point as a checkpoint.

We developed this tool based on the LLVM (Low Level Virtual Machine) [10] compiler suite. LLVM is an open-source compilation framework that uses an intermediate representation in Static Single Assignment (SSA) form. It has front ends to support multiple languages that are transformed into LLVM IR for analysis and optimization. It has been widely adopted in both academia and industry which is one of the motivations for our choice. One other interesting aspect of LLVM is its "Pass Framework" where most of the interesting parts of the compiler exist. Passes are used to perform analysis, transformations, and optimizations at the IR level. It also allows for extension or addition of a new analysis or optimization in a structured manner. We implemented our tool as LLVM passes. It has three main parts: the safe point identification phase, data optimization phase, and user-feedback phase. Figure 4 shows the architectural diagram of the tool.

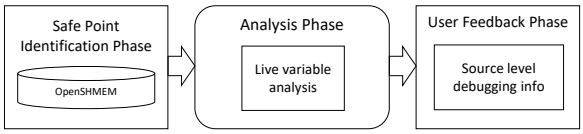

Fig. 4. Architectural Diagram of the tool

A. Safe Point Identification Phase

In this phase, we analyze the program to determine places where the data and computation are in a consistent and deterministic state (safe points). We utilize the analysis information described in Section III. Since the safe points are dependent on calls to a certain function or a series of functions that result in global synchronization, we track these function calls and retain this information for further data optimization analysis discussed next.

B. Data Optimization Phase

In this phase, we perform live variable analysis on the program. Based on the results, we determine which variables are live at a given safe point. During the analysis process, we treat arrays as a single entity i.e., if any element of an array is live at a certain point, the whole array is live. However, LLVM IR is in SSA form and every LLVM IR variable has exactly one definition. Therefore, the access to an element of an array may occur via multiple indirections and just from the instruction that actually accesses the element, we may not know which array it belongs to. To handle this problem, we track each of the SSA definitions back to its source.

We adapt live variable analysis to allow it to correctly deal with OpenSHMEM routines by furnishing definition and use information on the variables referenced in associated calls. In other words, we add partial awareness of the semantics of OpenSHMEM calls to the LLVM compiler. For example, `shmem_free` is used to free the symmetric memory allocated by OpenSHMEM-specific allocation routines. Although

`shmem_free` uses a pointer (pass by reference) argument passed to it to select which memory to free, it does not actually use the value of the elements allocated on those memories, nor would that value be used in the future (memory is already freed after this call). Therefore this is not a use of any variable and variables referenced in a `shmem_free` call should not, on the basis of this reference alone, be considered to be live. Without this information, the compiler would potentially be forced to extend the live range of the corresponding values to this point in the code.

C. User-feedback Phase

In this phase, we notify the user of the decisions made by the framework based on the data optimization phase. We provide information on where the safe points are and what data needs to be saved if a certain safe point is used as checkpoint. However, there is an additional problem. The analysis is performed by the tool at the IR level of the program where the source code level information (e.g., variable name, line number) is lost. In order to provide useful feedback to the user, we need to translate the IR-level information back to the source-level code, since the application developer using this will only have access to the latter. To solve this issue, we utilize the "LLVM source level debugging information" to:

- Find source-level positions (e.g., line numbers) for potential checkpoints
- Find source-level variable names and their position from IR level temporary variables

For this reason, this tool has to be run with the debug (-g) flag switched on and with no optimization enabled (optimization may result in source level information loss).

VI. Results and Analysis

In this section, we evaluate the safe point analysis and data optimization carried out by the compiler tool. We expect that the suggestions provided by this tool will be used by an application developer to create a fault-tolerant OpenSHMEM program by manually using a checkpointing library of their choice. We assume that the checkpointing library allows users to choose which data to save at each checkpoint i.e., has the ability to register or deregister variables to be checkpointed at each checkpoint. However, application-level checkpointing library support for OpenSHMEM is very scarce, and to the best of our knowledge, none exist at this point with the above-mentioned capability. We are in the process of adapting an existing library to serve this purpose.

Since there is no such library available for experimentation, for this work, we present the calculated result (instead of the execution result) that assumes such a library will be utilized. The goal of this work is to find safe points for checkpointing in an OpenSHMEM program and to optimize the data sets. Evaluation of the former does not require program execution, but rather a manual analysis to ascertain whether the points reported by the tool are indeed safe points and vice versa. With respect to the data optimization, the total amount of data used by a program and the size of the optimized data set computed via live variable analysis can be calculated accurately from the source code if the input parameters are known. Therefore, the results presented here should closely resemble the actual execution run using a checkpointing library.

We use 3 Benchmark applications, Transpose, Matrix Multiplication (MM), and Mandelbrot Set to evaluate this work. Due to the lack of benchmark applications written in OpenSHMEM, we developed two of the benchmarks (Transpose, and MM) used here.

We evaluate this tool based on two criteria: safe point identification and data optimization. For safe point identification, we validate the safe points identified by the tool via a manual investigation of the program by an expert. For data optimization, we compare the results of our optimization strategy against Hao et al. [36], which to the best of our knowledge, is the only prior work on OpenSHMEM application-level fault tolerance. Hao uses the "default" approach of saving all the available data during checkpointing. For a certain checkpoint, he saves all the data that are within the scope and potentially necessary for restart in any other checkpoints of the program. We use the term "All data" to refer to that result while using the term "lva-optimized" (Live variable analysis-optimized) to refer to the data optimization using our approach.

A. Transpose

The transposition is a very common communication pattern, which is commonly found in linear algebra calculations and FFT (Fast Fourier Transform) computations. This benchmark transposes a $N \times N$ matrix using a blocked approach. Each PE gets a sub-matrix containing a subset of rows, performs a local transpose, and then combines the results using an OpenSHMEM collective communication routine (`shmem_fcollect`). It has two functions, the `main` function, and the `transpose2D` function that does the transpose operation on the sub-matrix. We use the matrix size of 2048×2048, and 16 PEs for the experimental calculation.

Our tool is able to identify all 13 safe points that are present in the program; 6 of them are in `transpose2D`, and 7 of them are in `main`.

We compare the result of data optimization using our tool (lva-optimization) with Hao ("All data") in Figure 5. We show the result for all the safe points, although some of them may not be suitable for checkpointing (e.g., `shmem_init` and `shmem_finalize` are often the first and last parts of an OpensHMEM program; hence a checkpoint may not be necessary). We leave the choice of choosing checkpoints from these safe points up to the user.

In Figure 5, the X-axis shows the safe points in the program along with their source line number; the Y-axis shows the checkpoint data size at a specific safe point. We use normalized data sizes, normalized by "All data" (all the available data at that safe point). The blue bars (left) represent the amount of data to be checkpointed at a certain safe point using the "All data" approach, while the red bars (right) represent the amount of data to be checkpointed using our approach. We observe that our approach is able to optimize the amount of data to

be checkpointed significantly (checkpoints less than 1% of all available data) compared to the "All data" approach for some safe points, while requiring the same or close to the same data for others. For the cases where the amount of data to be checkpointed using our approach is negligible compared to the "All data" approach, the red bar (lva-optimized) is so small that it is not visible in the graph (e.g., `shmem_finalize` in the `main` function).

In the `main` function, we see that our approach does significantly better for the safe points that are at the latter part of the function. Often the memory (both symmetric and private) allocation and deallocation occur at the beginning and the end of a function, respectively, for better code readability. As a result, the allocated memory may be freed much later than its last use. This is potentially a problem for the "All data" checkpointing approach. Although the data is not used anymore, it is still part of the program memory; hence in the "All data" approach, this unused data must also be checkpointed. Our Live variable analysis approach can resolve this.

In the `transpose2D`, we see a slightly different behavior for the safe points in the latter part of the function. Although we see improvement using our approach, it is not as significant as we observed in the `main` function. This lack of improvement is because we do live variable analysis at the intra-procedural level; hence, we assume that any argument passed to a function by reference is "live" at the end of the function (the calling function may use updated values). In `transpose2D`, one of the arguments is a large array that is passed by reference. Since this variable can not be eliminated in our current approach, we see smaller gains than with the `main` function.

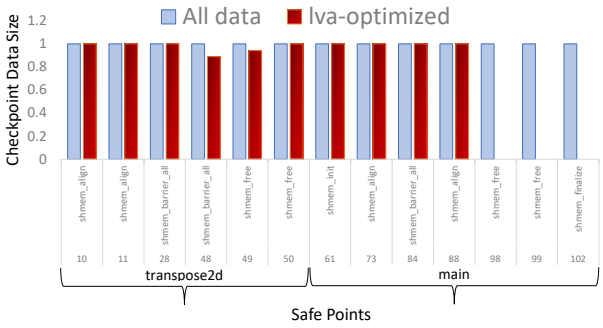

Fig. 5. Checkpoint data sizes at all the safe points in Transpose. Smaller is better (some bars for lva-optimized are not visible, because the value for lva-optimized is less than 1% of "All data").

B. Matrix Multiplication (MM)

This benchmark provides an OpenSHMEM implementation of Matrix Multiplication based on a variant of Cannon's algorithm. We use the matrix size of 2048×2048, and 16 PEs for the experimental calculation.

Figure 7 shows the comparison of our approach (lva-optimized) with the "All data" approach. The benchmark has 13 safe points, all of them in the `main` function. Most of the safe points are initialization/finalization and allocation/de-allocation routines while 3 of them are global barriers (`shmem_barrier_all`). Our tool is able to identify all of the safe points correctly.

In terms of data optimization, we observe similar results to the `main` function of Transpose; and for the same reason. Our approach is able to reduce the checkpoint data size significantly for the safe points in the latter part of the `main` function.

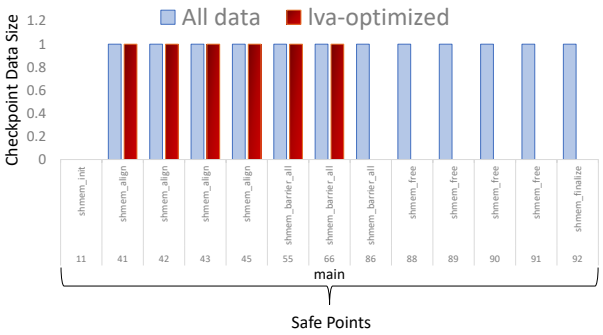

Fig. 6. Checkpoint data sizes at all the safe points in MM. Smaller is better (some bars for lva-optimized are not visible, because the value for lva-optimized is less than 1% compared to "All data").

C. Mandelbrot Set

This benchmark generates a greyscale image of the Mandelbrot set using the quadratic iteration function. The image is partitioned evenly across the PEs, and the computation is embarrassingly parallel. We use a 3200×3200 image of the Mandelbrot set, and 16 PEs for the experimental calculation.

The application has a total of 7 safe points, and our tool is able to identify all of them. 5 of the safe points are in the `draw_mandelbrot` function which does the actual computation, and 2 are in the `main` function which initializes/finalizes the OpensHMEM library and calls the `draw_mandelbrot` function. Therefore, the important safe points are in `draw_mandelbrot`, where all the data allocation, de-allocation, and computation occur. For data optimization, we see the familiar behavior already observed in Transpose and MM; we achieve significant savings using our approach at the latter safe points of the `draw_mandelbrot` function.

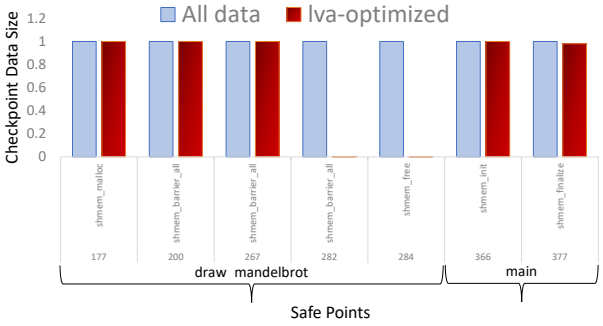

Fig. 7. Checkpoint data sizes at all the safe points in Mandelbrot. Smaller is better (some bars for lva-optimized are not visible, because the value for lva-optimized is less than 1% of "All data").

VII. Related Work

This work can trace its roots back to three overlapping research areas,

- Checkpoint-Restart Techniques
- Fault Tolerance in PGAS Programming Models
- Compiler and Tool Based Fault Tolerance

A. Checkpoint-Restart Techniques

Research on **Checkpoint-Restart Techniques** for Fault tolerance can be traced back to work performed as early as the 1960s by David Jasper [37]. Over the years, different checkpointing schemes such as uncoordinated, coordinated, communication-induced, incremental and multi-level checkpointing have been developed to meet the need of ever-changing computing environments and applications. Most of these schemes have been implemented to be used at the system level, at the application level, or both. In recent times many of these techniques have been developed or adapted for use in conjunction with Message-Passing Systems and Programming Models (e.g., MPI) as a result of their popularity in High Performance Computing. Surveys by Elnozahy et al. [25] and Dongara et al. [38] are excellent resources for information on these techniques.

B. Fault Tolerance in PGAS Programming Models

PGAS programming models such as OpenSHMEM are becoming popular in the HPC community due to their programmability, utilization of modern hardware, and performance affinity for applications with irregular communication patterns. However, consideration of fault tolerance for PGAS programming models is still scarce. Besta et al. [28] was one of the first to develop a generic model for reasoning about resilience in applications that use Remote Memory Access (RMA) and to introduce schemes for in-memory checkpointing and logging based protocols for them. Among other notable works that focus on specific PGAS programming models other than OpenSHMEM are [39] and [40]. In [39], Ellis et al. introduced a coordinated checkpointing protocol for UPC applications while in [40], Shahzad et al. developed tools to support fault tolerance in GASPI applications. The tools included a Health Check Library and a Fault-aware C/R library which in combination provide fault detection, propagation, and communication recovery.

Hao et al. [36], [41] was one of the first works to explore fault tolerance in the context of OpenSHMEM. Here the authors proposed an application-level checkpointing mechanism based on User Level Fault Mitigation (ULFM) where the shared global memory (symmetric memory) regions are replicated (backed up) across peer processes. However, it was up to the programmer to make sure that any data that may be necessary for restart is allocated in the symmetric memory. Since it was an application-level C/R scheme, the user code was responsible for managing the checkpoint and restart operations. Our work is designed to help application developers navigate the process of managing C/R operations and use schemes like this efficiently.

Garg et al. [42] introduced a different approach that utilized a system-level transparent checkpointing scheme for achieving OpenSHMEM fault tolerance. In their approach, they saved the checkpoints in stable storage which allowed them to save the computation to be used at a later time or on a different cluster. Despite these efforts, OpenSHMEM lacks an error model to provide proper error detection, propagation, and recovery. To address this issue, Bouteiller et al. [43] proposed such an error model as an extension to the OpenSHMEM API to solve these issues. However, so far this has not resulted in any modification to the specification. Compared to those works, ours is the first work to define and identify safe points for checkpointing in OpenSHMEM.

C. Compiler and Tool Based Fault Tolerance

Although application-level C/R can be the most effective C/R technique for overhead efficiency, doing it in a large-scale application can introduce huge implementation effort. Hence, researchers have looked into compilers and tools to ease this process via checkpointing suggestions or automatic checkpointing to recover from both soft [44] and hard failures [3]–[9].

Most of these works are based on source to source compilers and primarily focus on MPI programs. Porch [4] utilized a source to source compiler and user inputs (checkpointing routines and frequency) to insert checkpoint operations in a sequential C program. Bronevetsky et al. [5], [6], Yang et al. [7], and Rodriguez et al. [8] used source to source compilers to automatically insert checkpoints in MPI applications. In contrast to these efforts, our work focuses on OpenSHMEM, a PGAS programming model which introduces the added complexity of global consistency issues and utilizes an open source general-purpose compiler, LLVM. As a result, we are able to benefit from the on-going innovations in this rapidly evolving compiler infrastructure and plan to exploit the analyses already available in the LLVM infrastructure for future extensions of our tool. Rodriguez et al. [9] also provide an LLVM-based implementation that builds on top of their previous work [8], however it still targets MPI applications.

Another notable difference of our work from other research is that we do not perform automatic checkpointing, rather we focus on providing user feedback and facilitating checkpointing by the application developer. This is primarily due to the lack of checkpointing libraries in OpenSHMEM. Due to this scarcity, a user may have to modify or use application-based checkpointing libraries developed for other programming models (e.g., MPI). Hence, in this work we focus on helping the application developer use whatever library they choose. In the future, we plan to extend this work to do automatic checkpointing as well.

VIII. Conclusion and Future Work

In this work, we analyzed the OpenSHMEM programming model from a resilience perspective and defined "safe points" for inserting checkpointing library calls into OpenSHMEM applications. Moreover, we provide the set of data to be

saved at each potential checkpoint. We adapted live variable analysis for OpensHMEM to optimize the amount of data to be checkpointed at a certain safe point. We show that this optimization can sometimes result in a large improvement in the amount of data to be checkpointed (in some cases, less than "1%" of the available data is checkpointed).

Clearly, much needs to be done to provide comprehensive fault tolerance support for OpenSHMEM based on C/R. We are working to develop an application-level checkpointing library for OpenSHMEM that would enable application developers to directly exploit the results of our work. Moreover, we are developing additional techniques to further reduce the amount of checkpointed data, where feasible. In particular we are applying "memory exclusion" techniques that are based on memory read/write operations in OpenSHMEM programs. Finally, not all safe points are suitable for use as checkpoints. In this work, we left the decision of choosing which safe points should be used as checkpoints to the user. However, we plan to investigate ways to suggest suitable positions for checkpointing based on program analysis and the detected safe points.

ACKNOWLEDGMENT

This material is based upon work supported by the National Science Foundation under grant no. CCF-1725499. The authors would also like to thank Stony Brook Research Computing and Cyberinfrastructure, and the Institute for Advanced Computational Science at Stony Brook University for access to the HPC systems.

REFERENCES

[1] J. T. Daly *et al.*, "Application mttfe vs. platform mtbf: A fresh perspective on system reliability and application throughput for computations at scale," in *CCGRID 2008*.

[2] I. P. Egwutuoha *et al.*, "A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems," *The Journal of Supercomputing*, vol. 65, no. 3, pp. 1302–1326, 2013.

[3] C.-C. J. Li *et al.*, "Compiler-assisted full checkpointing," *Software: Practice and Experience*, 1994.

[4] B. Ramkumar *et al.*, "Portable checkpointing for heterogeneous architectures," in *FTCS 1997*.

[5] G. Bronevetsky *et al.*, "Automated application-level checkpointing of mpi programs," in *PPoPP 2003*.

[6] G. Bronevetsky *et al.*, "C 3: A system for automating application-level checkpointing of mpi programs," in *LCPC 2013*.

[7] X. Yang *et al.*, "Compiler-assisted application-level checkpointing for mpi programs," in *ICDCS 2008*.

[8] G. Rodríguez *et al.*, "Cppc: a compiler-assisted tool for portable checkpointing of message-passing applications," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 749–766, 2010.

[9] G. Rodríguez *et al.*, "Compiler-assisted checkpointing of parallel codes: The cetus and llvm experience," *International Journal of Parallel Programming*, vol. 41, no. 6, pp. 782–805, 2013.

[10] C. Lattner *et al.*, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[11] K. Yelick *et al.*, "Productivity and performance using partitioned global address space languages," in *Proceedings of the 2007 international workshop on Parallel symbolic computation*, 2007.

[12] T. S. Woodall *et al.*, "High performance rdma protocols in hpc," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, 2006.

[13] B. Chapman *et al.*, "Introducing openshmem: Shmem for the pgas community," in *PGAS 2010*.

[14] O. Community, "Openshmem application programming interface version 1.5," http://www.openshmem.org/, 2020.

[15] "Open source software solutions (osss) openshmem implementation on top of openucx (ucx) and pmix," https://github.com/openshmem-org/osss-ucx.

[16] "Mvapich2-x," http://mvapich.cse.ohio-state.edu/.

[17] "Sandia-openshmem (sos)," https://github.com/Sandia-OpenSHMEM/SOS.

[18] P. Shamis *et al.*, "Designing a high performance openshmem implementation using universal common communication substrate as a communication middleware," in *OpenSHMEM 2014*.

[19] "Openmpi," https://github.com/open-mpi/ompi.

[20] N. Namashivayam *et al.*, "Introducing cray openshmemx-a modular multi-communication layer openshmem implementation," in *OpenSHMEM 2018*.

[21] "Hpc-x™ openshmem," https://www.mellanox.com/products/hpc-x-software/hpc-x-openshmem.

[22] "Ibm spectrum™ mpi," https://www.ibm.com/products/spectrum-mpi.

[23] 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.

[24] J. Jose *et al.*, "Designing scalable graph500 benchmark with hybrid mpi+openshmem programming models," in *ISC 2013*.

[25] E. N. Elnozahy *et al.*, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys (CSUR)*, vol. 34, no. 3, pp. 375–408, 2002.

[26] R. Riesen *et al.*, "Alleviating scalability issues of checkpointing protocols," in *SC 2012*.

[27] E. N. Elnozahy *et al.*, "Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 2, pp. 97–108, 2004.

[28] M. Besta *et al.*, "Fault tolerance for remote memory access programming models," in *HPDC 2014*.

[29] P. H. Hargrove *et al.*, "Berkeley lab checkpoint/restart (blcr) for linux clusters," in *Journal of Physics: Conference Series*, vol. 46, no. 1, 2006, p. 494.

[30] L. Bautista-Gomez *et al.*, "Fti: high performance fault tolerance interface for hybrid systems," in *SC 2011*.

[31] F. Shahzad *et al.*, "Craft: A library for easier application-level checkpoint/restart and automatic fault tolerance," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 3, pp. 501–514, 2018.

[32] B. Nicolae *et al.*, "Veloc: Towards high performance adaptive asynchronous checkpointing at large scale," in *IPDPS 2019*.

[33] J. S. Plank *et al.*, "Compiler-assisted memory exclusion for fast checkpointing," *IEEE Technical Committee on Operating Systems and Application Environments*, vol. 7, no. 4, pp. 10–14, 1995.

[34] J. Heo *et al.*, "Space-efficient page-level incremental checkpointing," in *Proceedings of the 2005 ACM symposium on Applied computing*, 2005, pp. 1558–1562.

[35] J. S. Plank *et al.*, "Memory exclusion: Optimizing the performance of checkpointing systems," *Software: practice and experience*, vol. 29, no. 2, pp. 125–142, 1999.

[36] P. Hao *et al.*, "Check-pointing approach for fault tolerance in openshmem," in *OpenSHMEM 2014*.

[37] D. P. Jasper, "A discussion of checkpoint restart," *Software Age*, vol. 3, no. 10, pp. 9–14, 1969.

[38] J. Dongarra *et al.*, "Fault tolerance techniques for high-performance computing," in *Fault-Tolerance Techniques for High-Performance Computing*. Springer, 2015, pp. 3–85.

[39] M. Ellis *et al.*, "Fault tolerance for remote memory access in unified parallel c."

[40] F. Shahzad *et al.*, "Building and utilizing fault tolerance support tools for the gaspi applications," *The International Journal of High Performance Computing Applications*, vol. 32, no. 5, pp. 613–626, 2018.

[41] P. Hao *et al.*, "Fault tolerance for openshmem," in *PGAS 2014*.

[42] R. Garg *et al.*, "System-level transparent checkpointing for openshmem," in *OpenSHMEM 2016*.

[43] A. Bouteiller *et al.*, "Surviving errors with openshmem," in *OpenSHMEM 2016*.

[44] C. Chen *et al.*, "Care: compiler-assisted recovery from soft failures," in *SC 2019*.