MC-Checker: Detecting Memory Consistency Errors in MPI One-Sided Applications

Zhezhe Chen,[1⋆] James Dinan,[3] Zhen Tang,[2] Pavan Balaji,[4] Hua Zhong,[2] Jun Wei,[2] Tao Huang,[2] and Feng Qin[5]

[1]Twitter Inc.
zhezhec@twitter.com

[2]Inst. of Software, Chinese Academy of Sci.
Technology Center of Software Engineering
{*tangzhen12, zhongh, wj, tao*}*@otcaix.iscas.ac.cn*

[3]Intel Corporation
james.dinan@intel.com

[4]Argonne National Laboratory
Math. and Computer Science Division
balaji@mcs.anl.gov

[5]The Ohio State University
Dept. of Computer Science and Eng.
qin@cse.ohio-state.edu

***Abstract*—One-sided communication decouples data movement and synchronization by providing support for asynchronous reads and updates of distributed shared data. While such interfaces can be extremely efficient, they also impose challenges in properly performing asynchronous accesses to shared data.**

This paper presents MC-Checker, a new tool that detects memory consistency errors in MPI one-sided applications. MC-Checker first performs online instrumentation and captures relevant dynamic events, such as one-sided communications and load/store operations. MC-Checker then performs analysis to detect memory consistency errors. When found, errors are reported along with useful diagnostic information. Experiments indicate that MC-Checker is effective at detecting and diagnosing memory consistency bugs in MPI one-sided applications, with low overhead, ranging from 24.6% to 71.1%, with an average of 45.2%.

Categories and Subject Descriptors —
D.2.5 [*Testing and Debugging*]: Debugging Aids;
D.1.3 [*Concurrent Programming*]: Parallel programming;
D.4.4 [*Communications Management*]: Message sending

General Terms — Design, Performance, Reliability

Keywords — Bug Detection, MPI, One-Sided Communication

I. Introduction

MPI one-sided communication is becoming increasingly popular because it enables programmers to directly leverage the capabilities of RDMA interconnects. Compared with conventional two-sided communication, one-sided communication decouples data movement from synchronization, enabling greater concurrency in data movement and potentially greater scalability for applications. Furthermore, one-sided communication can provide higher efficiency by removing message matching and buffer coordination with the receiver, which occur with two-sided messaging.

The ability to decouple data movement from synchronization, while a strength of one-sided communication, also presents challenges to programmers. One must navigate a complex memory model and insert the synchronization operations needed to maintain data consistency in the presence of asynchronous and nonblocking data accesses from multiple processes. This complexity can expose applications to synchronization defects. Figure 1 shows such an example from a real-world application, where the one-sided `MPI_Get` operation (line 2) is nonblocking. As a result, the data may not be ready until the invocation of `MPI_Win_unlock` (line 6). This situation can cause the load access of `out` (line 3) to retrieve an old value and the store access of `out` (line 4) to be overwritten by a value retrieved from `MPI_Get` (line 2).

```
1: MPI_Win_lock(MPI_LOCK_EXCLUSIVE, 0, 0, win);
2: MPI_Get(&out, 1, MPI_INT, 0, 0, 1, MPI_INT, win);
3: if (out % 2 == 0) /* bug: load/store access of out */
4:     out++;
5: ...
6: MPI_Win_unlock(0, win);
```

Fig. 1: Real-world MPI one-sided communication bug.

Few tools exist to aid users of one-sided parallel programming models. Detection of memory model violations in MPI is further complicated by its multiple synchronization modes and operations, as well as its conservative semantics, that are designed to be portable across coherent and noncoherent hardware memory models. Significant work has been conducted to detect concurrency bugs in shared memory programs. Some approaches [1], [2], [3] detect data races via static analysis, while others [4], [5], [6], [7] leverage dynamic methods. However, these approaches cannot directly be applied to MPI programs that perform one-sided data access across distributed-memory platforms, because fundamentally different synchronization primitives are used. While numerous studies [8], [9], [10], [11], [12] have been done on bug detection in MPI programs, none have focused on memory consistency errors in MPI one-sided applications.

In this paper, we present a technique called MC-Checker to detect memory consistency errors in MPI one-sided applications. Such errors occur when two or more accesses to shared data—for example, MPI one-sided calls such as `MPI_Put` or local load/store accesses—conflict with each other, leading to an undefined or erroneous state. To detect memory consistency

⋆Z. Chen was with The Ohio State University when this work was conducted.

SC14, November 16-21, 2014, New Orleans, Louisiana, USA
978-1-4799-5500-8/14/$31.00

errors in MPI one-sided applications, we identify one-sided data access epochs and concurrent execution regions, and we check for memory model violations among the interleaving of concurrent accesses to shared data. We summarize the MPI memory model using compatibility tables that capture the memory model defined by the MPI specifications.If two operations conflict, MC-Checker will report the error and provide diagnostic information to help locate and fix the bug.

MC-Checker first performs static analysis to efficiently instrument CPU load and store operations that access shared data. Next, a runtime trace is gathered at each process to capture accesses to shared data. During offline analysis of this trace, MC-Checker constructs a dynamic data access directed acyclic graph (DAG), where events are ordered by the happens-before relation defined by the MPI one-sided memory model. Then it analyzes sets of operations that are unordered in the DAG, potentially spanning multiple processes, to detect consistency violations and data races that occurred, or could have occurred in the application execution. Based on these ideas, we have implemented a prototype of MC-Checker and evaluated it with three real-world and two injected bugs in five MPI one-sided applications. Our experiments show that MC-Checker can effectively detect memory consistency errors in the evaluated MPI applications.

In summary, MC-Checker has the following advantages:

1. MC-Checker is the first comprehensive approach to address memory consistency errors in MPI one-sided communication.

2. MC-Checker incurs low runtime overhead; we observe an average profiling runtime overhead of 45.2%. This low overhead is achieved by utilizing static analysis to instrument only relevant memory load/store accesses.

3. MC-Checker is easy to use and requires no program modifications. Although our current implementation focuses on the MPI applications written in C, we find no difficulties in extending MC-Checker to MPI applications written in other languages such as Fortran.

4. The analysis techniques used by MC-Checker can also be applied to other one-sided programming models. Most one-sided programming models share similar ordering and data consistency semantics. Our analysis can be easily applied to those programming models by defining the set of consistency rules and ordering relations needed to build and analyze the data access DAG.

II. Memory Consistency Errors in MPI RMA

A memory consistency error in MPI is an erroneous program execution state, defined by the MPI semantics [13], that results in undefined values for buffers exposed in an MPI remote memory access (RMA) window. It occurs when at least two conflicting operations are executed concurrently at run time. In this work, we focus on the one-sided communication model defined in the MPI 2.2 specification [13], for simplicity. We discuss applying our techniques to new MPI 3.0 standard [14] in Section V.

A. MPI Remote Memory Access Model

The MPI 2.2 standard defines a highly portable memory model for one-sided communication. This model allows applications to be portable even to systems without coherent memory subsystems. On such systems, the MPI implementation must be able to perform software coherence. To this end, MPI defines its memory model in terms of conflicting operations that make it impossible for a software management mechanism to resolve a consistent result in one or more locations in memory that are exposed for remote access.

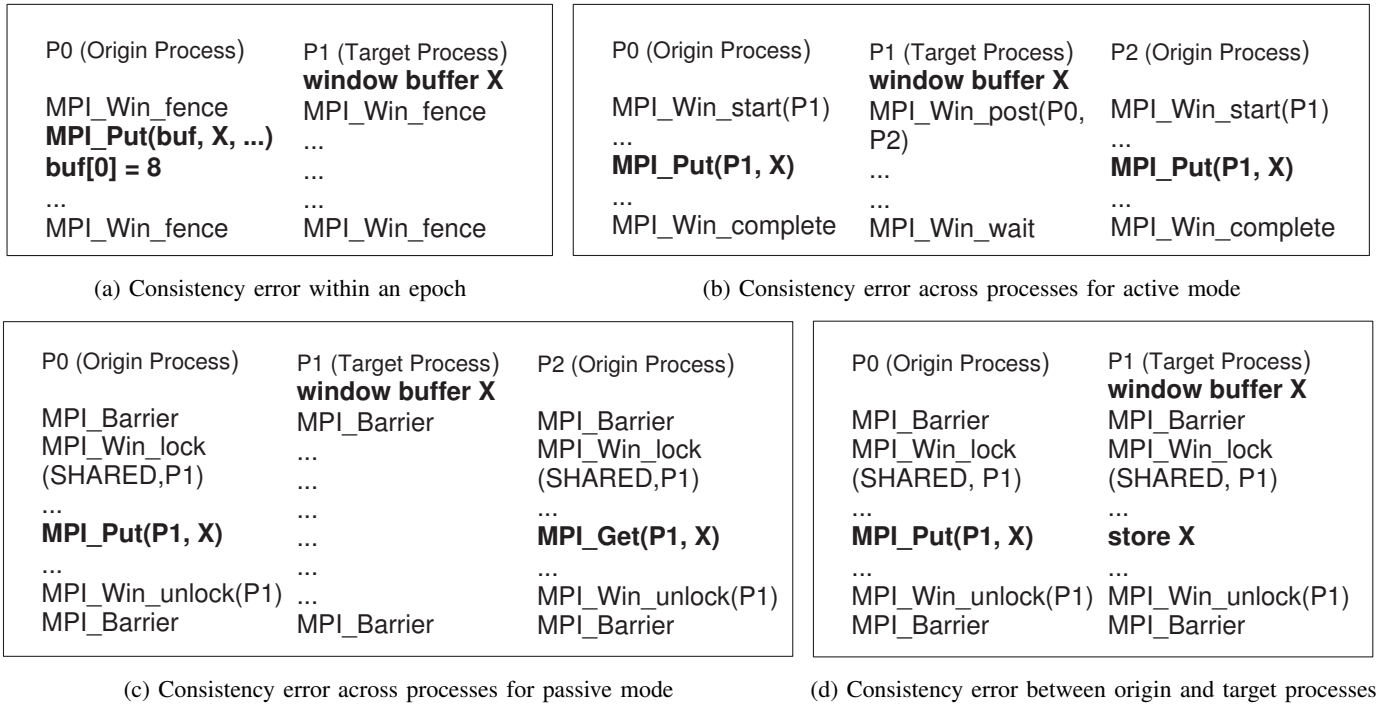
TABLE I: Compatibility matrix of RMA operations. BOTH indicates that both overlapping and nonoverlapping combinations of the given operations are permitted; NON-OV indicates that only non-overlapping combinations are permitted; and ERROR indicates that the combination is erroneous.

	Load	Store	Get	Put	Acc
Load	BOTH	BOTH	BOTH	NON-OV	NON-OV
Store	BOTH	BOTH	NON-OV	ERROR	ERROR
Get	BOTH	NON-OV	BOTH	NON-OV	NON-OV
Put	NON-OV	ERROR	NON-OV	NON-OV	NON-OV
Acc	NON-OV	ERROR	NON-OV	NON-OV	BOTH

MPI's put, get, and accumulate one-sided communication operations are nonblocking and are grouped into completion units, called epochs. An epoch is a program execution region that starts with an RMA synchronization operation (e.g., `MPI_Win_fence` or `MPI_Win_lock`) and ends with a matching RMA synchronization operation (e.g., `MPI_Win_fence` or `MPI_Win_unlock`). Operations within the same epoch can conflict with each other because they are nonblocking and the order in which they are applied is undefined. Two memory operations a and b within an epoch are called conflicting if they access overlapping local memory locations and one is an update operation. An exception to this rule is made for accumulate operations that use the same operation and basic datatype. Conflicts between epochs originating from different processes also occur when the epochs occur concurrently on overlapping regions at the same target process. At least one operation must be a write, and the same exception is made for accumulate operations. Table I shows which combinations are valid.

B. Memory Consistency Errors

Two memory operations a and b (i.e., remote or local memory accesses) in MPI programs are concurrent if there are no happens-before relation [15] and consistency ordering [13] between them. A happens-before relation between two operations a and b (i.e., $a \xrightarrow{hb} b$) can be either the program order within one process (i.e., the previous instruction is executed before the later instruction) or the synchronization order between different processes (e.g., `MPI_Send` at the source process completes before `MPI_Recv` at the destination process). A consistency order between two operations a and b (i.e., $a \xrightarrow{co} b$) guarantees that the memory effects of a are visible before b. This order is necessary because some synchronization actions (e.g., `MPI_Win_lock/unlock`) order memory accesses without synchronizing processes. For example, if a is nonblocking, b is the operation immediately following a, and both a and b access overlapping buffers, there is no consistency order between a and b because of a's nonblocking nature.



(a) Consistency error within an epoch

(b) Consistency error across processes for active mode

(c) Consistency error across processes for passive mode

(d) Consistency error between origin and target processes

Fig. 2: Examples of memory consistency errors. Bold operations are unordered, resulting in conflicts.

Figure 2a shows an example of memory consistency errors that occur within the same epoch. `MPI_Put` sends data in `buf` from process P0 to process P1. After `MPI_Put`, the data in `buf` may or may not be sent out because the `MPI_Put` is nonblocking. However, the data may be corrupted by the store operation right after `MPI_Put`. Such errors are common in applications using one-sided communication. For example, an older version of the Asynchronous Dynamic Load Balancing (ADLB) [16] library, which is used in the Green's function Monte Carlo (GFMC) [17] nuclear physics application, used `MPI_Put` to transfer data from a stack variable in a function and returned from the function without waiting for the completion of that operation, since the epoch was closed later elsewhere in the program. This procedure worked correctly for several years on multiple generations of machines since on most platforms small variables are copied into internal temporary communication buffers for later transmission. When the code was ported to the IBM Blue Gene/Q in early 2012, however, in some cases the MPI implementation would run out of internal temporary buffers and would need to transmit the data later. In such cases, the function stack was overwritten by other functions, resulting in data corruption[18].

Figure 2b shows an example of memory consistency errors across processes in an active target epoch. The `MPI_Put` operations in P0 and P2 are conflicting because they may access the window location in P1 concurrently, leading to data corruption or undefined results during program execution. Figure 2c shows a similar example of memory consistency errors across processes in a passive target epoch. Figure 2d shows another example of memory consistency errors where the `MPI_Put` in the origin process conflicts with the store operation in the target process because they will write to the same buffer concurrently and may cause data corruption.

III. Detecting Memory Consistency Errors

MC-Checker performs three main steps to detect memory consistency errors occurring in an MPI program. First, MC-Checker collects relevant runtime events in all running MPI processes. Next, MC-Checker converts the traces to a DAG offline, by applying a happens-before relation to the collected runtime events, and extracts concurrent regions from the DAG. MC-Checker then applies a data access ruleset to operations that are unordered in the DAG in order to detect memory model violations. While we have focused on the MPI 2.2 one-sided communication model in this work, any one-sided or PGAS model for which a DAG based on happens-before relation and data access ruleset can be constructed is amenable to analysis by MC-Checker.

A. Profiling Runtime Events

The first step in MC-Checker analysis is online collection of an event trace from a running MPI program. MC-Checker captures load/store and RMA accesses to data in an RMA window, MPI RMA synchronization operations, and MPI operations that synchronize among the processes (e.g., send/recv or barrier). Interprocess synchronization events must be captured because they can partially order local or remote memory accesses. As a result, MC-Checker can avoid establishing a more detailed ordering and eliminate false positives. For example, two well-synchronized memory accesses from different processes will never conflict if they are separated by a call to `MPI_Barrier`.

B. Constructing DAG and Concurrent Regions

After collecting the relevant runtime events for each MPI process, MC-Checker converts these runtime events to a DAG.

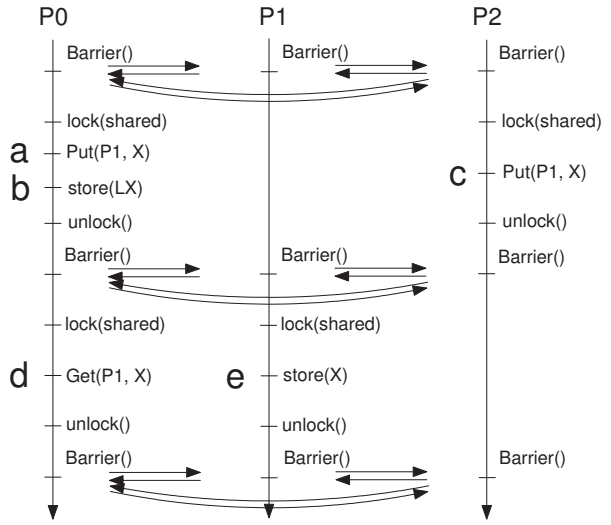


Fig. 3: Execution timeline of a representative bug example. We abbreviate the names of MPI operations by omitting the "MPI" or "MPI_Win" prefix. LX in operation b is the local buffer of operation a.

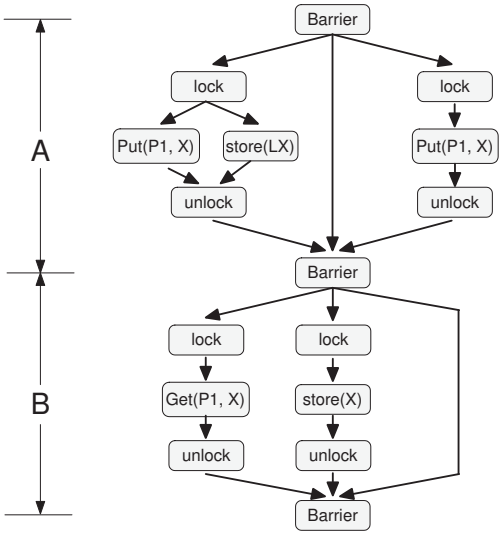


Fig. 4: Data access DAG for the execution trace shown in Figure 3.

Figure 3 shows an example of the execution of an MPI program with three MPI processes and the relevant runtime events that are collected. Figure 4 shows the resulting DAG for regions A and B where the happens-before relation has been applied to the event trace gathered in Figure 3. In this DAG, concurrent put and store operations update the same memory location, resulting in a data race.

The first step in transforming the event trace to a DAG is to represent each runtime event as a vertex. Each vertex is labeled with the process ID that performed the operation and its parameters, such as memory access type, memory locations accessed, and target process ID. Next, MC-Checker connects vertices within each process based on the happens-before relation. MPI RMA accesses are grouped into epochs, which are program execution regions that start with an RMA synchronization operation (e.g., `MPI_Win_fence`) and end with a matching RMA synchronization operation (e.g., `MPI_Win_fence`). Nonblocking RMA operations occur after the RMA synchronization operation that begins their epoch. While the epochs in each MPI process are ordered based on their execution, the nonblocking RMA operations within each epoch are not ordered.

Next, MC-Checker connects the vertices that are ordered across processes. Specifically, MC-Checker matches all synchronization and communication events across different processes, such as MPI collective calls and blocking send/receive calls. For each pair of such matched events, MC-Checker connects the corresponding pair of vertices by a directed edge according to the happens-before relation. For example, MC-Checker forms a directed edge between the paired `MPI_Send` and `MPI_Recv`. Similarly, it constructs two directed edges between a pair of `MPI_Barrier`'s in two different processes. As a result of this step, a DAG is constructed by MC-Checker.

MC-Checker then extracts unordered subgraphs, or *concurrent regions* from the data access DAG, corresponding to sets of operations that could have occurred concurrently in the running program. In Figure 4 we show two concurrent regions, A and B. While analyzing the DAG, MC-Checker identifies global synchronization events (e.g., via barrier operations) that partition the DAG. These synchronization events essentially truncate the DAG into multiple execution regions, which are sequentially ordered and can be used to improve the efficiency of the analysis.

C. Identifying Conflicting Operations

For each concurrent region, MC-Checker applies a memory model ruleset to detect memory consistency errors. There are two classes of errors: errors caused by conflicting operations within an epoch at a single process and errors caused by conflicting operations across multiple processes.

For each concurrent region, MC-Checker first scans all the vertices belonging to a process and identifies all the epochs within the process by matching the synchronization calls. For each identified epoch, MC-Checker further checks memory access operations against a memory model ruleset. The ruleset defines the memory operations, such as local memory read/write accesses and one-sided communication calls, that can be executed concurrently. Such compatibility tables may be different for different one-sided communication or PGAS models.

Detecting conflicting operations across processes is more complex. For each concurrent region, MC-Checker extracts all the vertices and edges of each pair of processes and performs a compatibility check for the extracted vertices between the two processes. Similarly the compatibility tables can be different for different one-sided communication or PGAS models.

If a pair of conflicting operations is found for either of the two types of memory errors, MC-Checker will report the memory consistency error and provide diagnostic information. For example, in Figure 2a, MC-Checker will detect the conflicting pair of operations `MPI_Put` and local write operation to the buffer within one epoch and report the

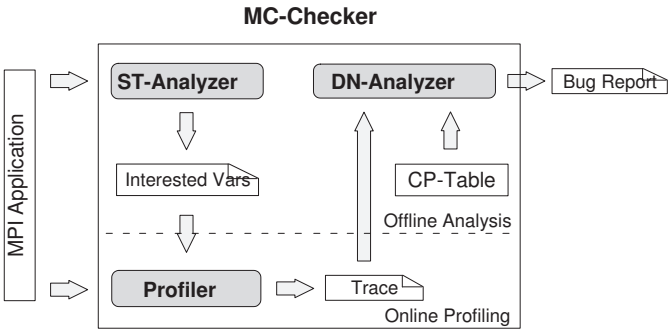


Fig. 5: Design overview of MC-Checker.

error with relevant runtime events. In Figure 2b, MC-Checker will identify the concurrent regions between P0 and P2. For this pair of concurrent regions, MC-Checker will detect the conflicting `MPI_Put` operations in P0 and P2 and report this error. Similarly, in Figure 2c, MC-Checker will identify the concurrent regions and detect the conflicting `MPI_Put` and `MPI_Get` in P0 and P1, respectively.

We note that the passive target mode in MPI one-sided communication requires other MPI calls such as `MPI_Barrier` to perform interprocess synchronization. Conflicting operations in Figure 2b and 2c occur between the origin processes. Conflicting operations may also occur between the origin and target processes, as shown in Figure 2d. In this example, MC-Checker will detect that the `MPI_Put` by P0 and the store operation by P1 conflict with each other. After detecting the conflicting operations, MC-Checker will provide diagnostic information, such as pairs of conflicting operations and operation locations including file names, routine names, and line numbers, to help programmers locate and fix the bugs.

IV. Design of MC-Checker and Implementation

MC-Checker consists of three components: ST-Analyzer, Profiler, and DN-Analyzer. As shown in Figure 5, ST-Analyzer performs static analysis of MPI applications at the source code level and generates a report consisting of relevant variable names that need to be monitored for their load/store instructions. Based on the result of ST-Analyzer, Profiler then instruments MPI applications before program execution and logs relevant runtime events to the trace files during program execution. DN-Analyzer analyzes the runtime traces and detects memory consistency errors. If it identifies a pair of conflicting operations that are not ordered, DN-Analyzer will report the error and provide important diagnostic information to help locate and fix the bug. Of these three components, Profiler is an online component because the instrumented code is executed together with the MPI application; the other two are offline components.

A. ST-Analyzer: Identifying Relevant Memory Accesses

To perform thorough memory consistency checking, Profiler needs to instrument every memory access, leading to large runtime overhead. To address this issue, ST-Analyzer performs static analysis to identify relevant memory access instructions that need to be instrumented by Profiler.

In order to identify relevant memory access instructions for detecting memory consistency errors, one straightforward approach is to analyze each load/store instruction, branch, loop, the scope of each variable, and so on. Such complete and sound analysis is expensive and slow, however, since it requires context-sensitive and parameter-sensitive interprocedure analysis and sound pointer alias analysis.

ST-Analyzer simplifies the analysis without losing the load/store instructions in which we are interested. First, ST-Analyzer identifies all variables that belong to the window buffers or the buffers being accessed by one-sided communication calls. It labels these variables as "relevant." Then ST-Analyzer propagates such labels by following pointer assignments or function calls involving pointers. After that, ST-Analyzer records all labeled variables in the report. These variables are the ones we would like to instrument in the load/store instructions. Our design of ST-Analyzer is conservative in that it is insensitive to branch and loop. In other words, ST-Analyzer may mark some variables that do not need to be instrumented in reality, but it will not fail to mark those that need to be instrumented. The current prototype of ST-Analyzer is implemented by using the Clang front-end as a library in the LLVM [19] compiler framework.

B. Profiler: Collecting Runtime Information

To facilitate error checking, Profiler collects relevant runtime events and logs them in trace files. In particular, we need to collect four types of MPI calls during program execution. The first type is MPI one-sided calls, including MPI one-sided initialization calls, communication calls, and synchronization calls. Examples of this type include `MPI_Win_create`, `MPI_Put`, and `MPI_Win_fence`. These MPI calls are required because they are directly related to one-sided communication. The second type is MPI datatype manipulation routines such as `MPI_Type_struct`. These routines create new datatypes from existing primitive datatypes (e.g., `MPI_INT`) or other user-defined datatypes. They are needed because message data during one-sided communication may reside in a memory location specified by such derived datatypes. The third type is general synchronization calls. Examples include `MPI_Barrier`, `MPI_Bcast`, `MPI_Send`, and `MPI_Recv`. These synchronization calls should be instrumented because they may affect data availability for MPI communication and the derivation of concurrency among operations. The fourth type is MPI support routines such as `MPI_Comm_rank` and `MPI_Group_incl`. We need them to retrieve basic information for error detection.

In addition to instrumenting the four types of MPI calls, Profiler needs to collect memory load/store accesses because they may conflict with one-sided communication operations. Profiler takes the report from ST-Analyzer as its input. The report contains the names of the variables and pointers that need to be instrumented for their memory accesses. Profiler logs the memory accesses of the identified variables to the trace files.

For each instrumented function call, Profiler logs the function name and the arguments to the trace files. For each memory access, Profiler logs the access type (e.g., read or write), the address of the accessed memory, and the size of the

accessed memory data. Such information is sufficient to detect memory consistency errors for MPI one-sided communication.

We leverage techniques from the LLVM [19] compiler framework to implement the current prototype of Profiler. Specifically, the LLVM compiler front-end first transforms the MPI application to LLVM intermediate representation (IR). Next, the Profiler pass is applied and transforms the original IR into the instrumented IR. Then, the code generation part of LLVM converts the instrumented IR into binary code.

C. DN-Analyzer: Trace Analysis and Bug Detection

DN-Analyzer preprocesses the collected traces, constructs DAG and concurrent regions, and then detects memory consistency errors offline.

1) Preprocessing trace files and extract information: Before checking memory consistency errors, DN-Analyzer preprocesses the trace files to retrieve some basic information. There are four types of such information: communicator, group, window buffer, and datatype.

a) Processing communicators and groups: An MPI program can use communicator/group manipulating routines to create user-defined communicators/groups based on previously defined or basic communicators/groups. For example, the MPI call `MPI_Group_incl` creates a new group including part of the processes from an old group. The rank information in this function call is relative to the old group, not the basic group associated with `MPI_COMM_WORLD`. For convenience, DN-Analyzer transfers all relative ranks in the newly created communicators/groups to absolute ranks in the basic communicator. DN-Analyzer then stores all the communicator/group information in a hash map for future analysis.

b) Processing window buffers: A window buffer is created by the MPI call `MPI_Win_create`. After finishing the call, an MPI program generates a handle to represent this chunk of window buffer for one-sided communication. DN-Analyzer stores the handle of the window buffer in the hash map. As a result, when one-sided communication involves the window buffer, DN-Analyzer can retrieve its detailed information for error detection.

c) Processing datatypes: MPI datatypes are complex. They range from simple primitive contiguous datatypes such as `MPI_INT` to complex user-defined noncontiguous datatypes. DN-Analyzer uses a *data-map* structure to represent a datatype. A data-map consists of a series of segments, each containing the displacement and the length of a contiguous chunk of the buffer. For example, `MPI_INT` is represented as a data-map $\{(0, 4)\}$, where 0 is displacement and 4 is length. Similarly, a derived datatype containing two `MPI_INT`s separated by an 8-byte gap is represented as $\{(0, 4), (12, 4)\}$. DN-Analyzer processes all datatype-manipulating routines and stores them in the hash map.

2) Constructing DAG and concurrent regions: As shown in Figure 2, MPI one-sided communication calls may cause memory consistency errors with other one-sided communication calls or load/store operations in another process. However, not every pair of operations will cause such errors. The reason is that MPI applications have synchronization calls to enforce happens-before [15] and/or consistency ordering relations between two operations. For example, in Figure 3, operation c (i.e., `MPI_Put` in process P2) and d (i.e., `MPI_Get` in process P1) will not cause memory consistency errors since the barriers in P0, P1, and P2 make c always happens before d (i.e., $c \xrightarrow{hb} d$) and c is consistent before d (i.e., $c \xrightarrow{co} d$). Only when two operations fall into a concurrent execution region may they cause memory consistency errors, such as a and c or d and e in this example. In this paper, a concurrent program region is defined as a group of program regions across multiple processes that can be executed concurrently, without happens-before and consistency ordering relations. Since the concurrent regions are formed by MPI synchronization calls, DN-Analyzer needs only to identify matching MPI synchronization calls across different processes and detect conflicting operations in each concurrent region.

a) Matching synchronization calls: To identify the concurrent regions, DN-Analyzer first matches MPI synchronization calls across multiple processes, which form happens-before and consistency ordering relations among processes. A concurrent region is formed by the set of program regions in all processes that are not ordered by happens-before and consistency ordering relations.

MPI has different types of synchronization calls, including all collective calls, blocking send/receive, and nonblocking send with its corresponding wait or test in the nonblocking receive process. Matching all these synchronization calls is a challenging task. A straightforward approach is as follows. For each synchronization call, one scans through all the traces in the corresponding processes and locates its matching synchronization calls. This algorithm is time-consuming and error-prone, however, especially for large trace files. The reason is that MPI applications may use the same synchronization calls with the same arguments many times during program execution. Hence, it is not easy to determine which synchronization calls to match in other processes for a specific synchronization call in the current process.

DN-Analyzer uses a more efficient approach for matching synchronization calls. This approach tries to simulate the progress of real MPI processes. Specifically, DN-Analyzer maintains a vector of "progress counters" to track the matching progress for each process. The progress counter for a process is defined as the ratio of the number of matched entries over the number of all entries in the process. Each entry is a runtime event logged in the trace file for a process. At each step, DN-Analyzer selects the process counter with the minimum value and starts the matching process for its first unmatched entry. If the entry is not a synchronization call, DN-Analyzer will skip it and update the progress counter of the current process. If the entry is a synchronization call, DN-Analyzer retrieves the argument values of the synchronization call. Based on the argument information, DN-Analyzer figures out other processes that have matching synchronization calls. For collective calls, DN-Analyzer can retrieve this information from the communicators. For blocking send/receive, the information can be fetched from send/receive arguments that specify the corresponding receive/send rank. After identifying the target processes of the matching synchronization calls, DN-Analyzer handles one process at a time. For each matching process, DN-Analyzer does not search from the beginning of

Algorithm 1 Match synchronization calls

```
1: minProgRank ← 0
2: while progress[minProgRank] < 1 do
3:    entry ← getEntry(progress[minProgRank])
4:    if entry is not synchronization call then
5:       continue
6:    else
7:       match synchronization calls with other processes and
         store matching information
8:    end if
9:    update progress[minProgRank]
10:   minProgRank ← getMinProgRank(progress)
11: end while
```

the trace file since doing so is inefficient and unnecessary. Instead, DN-Analyzer locates the first unmatched entry from recorded progress counter information and searches for the nearest matching synchronization calls from there. After a matching call is found, DN-Analyzer stores the matching information for both synchronization calls.

Algorithm 1 shows how DN-Analyzer identifies matching synchronization calls. Line 1 initializes the `minProgRank` to 0. The while loop in line 2 checks whether the matching completes for a particular process. Inside the while loop, DN-Analyzer retrieves the first unmatched entry for the process in line 3. If it is not a synchronization call, it simply skips the entry in line 5. Otherwise, DN-Analyzer finds matching calls in other processes and stores the matching information in line 7. Then it updates the progress for the current rank in line 9 and finds the next minimum progress rank in line 10.

b) Representing concurrent regions using DAG: After matching the synchronization calls, DN-Analyzer leverages the DAG to represent the synchronization information. The happens-before relation is represented by a direct edge. As shown in Figure 4, the `MPI_Barrier` is the synchronization call, which can be represented as one node in the DAG. If the synchronization calls are blocking send and receive, they will be represented by a direct edge pointing from send to receive. In this DAG, we can also get two concurrent regions, A and B.

3) Detecting conflicting operations within an epoch: As Figure 2a shows, MPI applications with one-sided communication can have conflicting operations within an epoch. The reason is that these one-sided operations are nonblocking. The data may not be moved in/out until the epoch ends. Based on the MPI specification [13], we derived the operation compatibility rules for operations within an epoch: (a) if the `MPI_Get` is followed by any other operation, only non-overlapping accesses are permitted; (b) if `MPI_Put/Acc` is followed by local store or `MPI_Get`, only non-overlapping accesses are permitted; and (c) for all other cases, both overlapping and non-overlapping accesses are permitted. We can detect conflicting operations based on these rules. As shown in Figure 3, operations *a* and *b* are conflicting operations based on rule b because they access overlapping buffer.

When we are constructing the DAG, the nonblocking operations are represented parallel with other following operations. As shown in Figure 4, `MPI_Put` is parallel with store. For all these concurrent operations within an epoch, we will check the rules for potential violations. If a violation is found, we will report the bug and provide diagnostic information.

4) Detecting conflicting operations across processes: After identifying the concurrent regions, DN-Analyzer detects possible memory consistency errors for each concurrent region. A straightforward method involves DN-Analyzer examining each pair of operations in a concurrent region against the compatibility table. Unfortunately, the time complexity is combinatorial with respect to the total number of operations within one concurrent region. Can we do better?

By further analyzing MPI specification and programs, we observe that memory consistency errors across processes can occur only in the window buffers at target processes. There are two types of memory consistency errors across processes. The first type involves two operations, both of which are MPI one-sided communication calls, such as the example in Figure 2b and 2c. One condition for such errors is that they perform remote memory operations on the same target process. The second type involves one MPI one-sided communication call and one local load/store, such as the example in Figure 2d. The local load/store is performed in the target process of the one-sided communication calls.

Based on this observation, we devise a more efficient error detection approach whose time complexity is linear to the total number of operations. In particular, we need to examine the memory operations only on the window buffer in the target process. This approach has two steps. First, DN-Analyzer checks the memory consistency errors for all one-sided MPI calls that operate on the window buffers in the target processes. Additionally, it stores all one-sided MPI operations on the window buffers in the target processes for the next step. Second, DN-Analyzer checks whether the local memory operations conflict with the stored remote one-sided operations for each window buffer. The local operations include local load/store and the MPI calls that access a local buffer. We discuss more technical details of this approach as follows.

For each concurrent region, DN-Analyzer uses a vector to store the information of the window buffers, one buffer per vector entry. Each entry stores the rank of the target process, the window location, and all previous one-sided operations one the window buffer. First, DN-Analyzer scans all of the remote one-sided operations in the concurrent region and record them to the correspond vector entry. Before recording them, DN-Analyzer need to check the compatibility with existing operations to the window buffer against Table I. If a conflict is found, DN-Analyzer reports the error and provide relevant diagnostic information.

After checking the memory consistency errors for all one-sided calls, DN-Analyzer examines all local operations for each process to see whether they conflict with any previous remote one-sided calls to the overlapping window buffers stored in the vector. The local operations include the local load/store and all MPI calls performed to a local buffer. Since `MPI_Put` and `MPI_Get` access a local buffer, they can be treated as local load and store, respectively.

We note that if a local memory operation under examination is a local store (not `MPI_Get` accessing a local buffer), we need to treat it differently. The reason is that MPI 2.2 specifies

that a local store cannot be combined with any `MPI_Put` or `MPI_Accumulate` even when they do not have any buffer overlap. DN-Analyzer then checks whether there are buffer overlaps between the local buffer currently under examination and the window buffers for the process.

V. Discussion

Several factors can complicate the detection of memory consistency errors. Invalid arguments to MPI operations can lead to erroneous programs. MC-Checker does not check for these errors; instead it relies on the MPI implementation or on another tool, such as Marmot [10], to detect such errors. Pointer aliasing is also a challenge in capturing direct load/store access to the window buffer. ST-Analyzer can detect most pointer aliasing by detecting pointer assignments and passing pointers to function calls. However, there are other mechanisms by which pointers can be aliased, such as through memory copies. Currently, pointer aliasing is a source for potential false negatives. ST-Analyzer could be extended to track such operations and catch the additional sources of pointer aliasing through dynamic tracking.

DN-Analyzer currently captures direct process-to-process synchronization; however, indirect synchronization, for example through send and receive operations by several different processes, can result in a transitive ordering. In addition, several sets of synchronizing MPI operations are omitted from our analysis, such as collectives and nonblocking operations. Currently, the lack of such synchronizations is a potential source of false positives. We limit the scope of analysis to reduce overheads; to capture such indirect synchronizations in our analysis, we would need to perform a complete analysis after building the DAG.

In this work, we have focused on the MPI-2 memory model. This memory model represents a subset of the MPI-3 functionality, and operating within this constrained environment has enabled us to clearly identify the type of analysis needed to detect RMA data races. We believe that the techniques we have developed can be applied to the MPI-3 one-sided communication model, as well as to other one-sided and PGAS models. To extend this analysis to a new model, one must define a happens-before relation that orders operations on remotely accessible data and can be used to create a data access DAG. Also needed is a set of rules that define combinations of operations that result in undefined or erroneous behavior when performed concurrently.

VI. Evaluation Methodology

We perform our experiments on the Glenn cluster at the Ohio Supercomputer Center [20]. The cluster contains 658 computer nodes. Each node is a quad-core machine with 2.5 GHz AMD® Opteron⋆ CPU, 24 GB RAM, and 393 GB local disk space. The operating system running on the cluster is Linux⋆ 2.6.18. We use the cluster for online profiling. For analyzing the trace files, we use a computer with 2.67 GHz Intel® Core™ i5 processor, 4 GB RAM, and 1 TB hard drive. The static analyzer and online profiler are implemented by using the LLVM compiler framework [19]. The offline analyzer is implemented as a single-threaded application using C++. We plan to further improve it by using multithreaded programming.

We evaluate the effectiveness of MC-Checker by using five real-world MPI one-sided applications: (1) emulate, a program emulating distributed shared memory; (2) BT-broadcast [21], a binary tree broadcast algorithm using one-sided MPI discussed in a paper's appendix; (3) lockopts, an RMA test case in the MPICH [22] library package; (4) ping-pong, a benchmark using ARMCI-MPI [23] in the MPICH library package; and (5) jacobi, an MPI one-sided implementation of the Jacobi method. The first three applications contain real-world bug cases of memory consistency errors within an epoch or across processes. We change exclusive lock to shared lock for the lockopts bug. To evaluate whether our tool can detect most of the buggy scenarios, we inject another two memory consistency errors in the last two applications.

We evaluate the runtime overhead of MC-Checker by using three applications in the GA [24] package (Lennard-Jones, SCF, and Boltzmann), SKaMPI [25], and LU in the NAS Parallel Benchmarks. We replace the ARMCI library with ARMCI-MPI [23] so that GA will use ARMCI-MPI as communication library. The ARMCI-MPI library is implemented by using MPI one-sided communication and is available in MPICH-3.0.4 package. LU is run in the configuration of 1500 by 1500 matrix. To evaluate the runtime overhead, we run the applications with two configurations: one with MC-Checker's Profiler and one without MC-Checker's Profiler. Each configuration is run five times in 64 processes. We also evaluate the scalability of MC-Checker's Profiler on the LU benchmark with various numbers of processes ranging from 8 to 128.

VII. Experimental Results

We measure the overall effectiveness of MC-Checker by examining whether it can detect the bugs and locate their root causes. The results are shown in Table II. Also listed is the bug information including error location, root cause, and failure symptom. We record the number of processes involved in our experiments for each bug case.

As indicated in the table, MC-Checker not only detects all the evaluated three real-world and two injected bugs but also pinpoints the root causes of all five bugs. Additionally, MC-Checker detects the bugs residing in different error locations, with root causes of different conflicting operations and failure symptoms. For example, MC-Checker detects the errors in emulate, where the bug resides within an epoch and is caused by conflicting `MPI_Get` and local load/store operations. As another example, MC-Checker locates the root cause of the bug in lockopts where the bug occurs across two processes due to conflicting local load/store and remote `MPI_Put/Get` operations. MC-Checker is effective in detecting memory consistency errors because it can accurately capture the timing of the MPI one-sided calls and the local load/store operations; then it identifies the conflicting operations.

Table II also shows that MC-Checker's detection capability is not affected by the scale of the system. MC-Checker can detect the memory consistency errors occurring in a small scale such as emulate in 2 processes, as well as the errors occurring in a larger scale such as lockopts in 64 processes. The reason is that MC-Checker is a rule-based approach. It can accurately capture program semantics at runtime and detect violation of

TABLE II: Overall effectiveness of MC-Checker. Note that some bug IDs use the date of the applications and some use svn revision or version number. The bug whose application name ended with "rev" is revised. The bug ended with "inj" is injected.

MPI Apps	Bug IDs	Detect?	Pinpoint Root Cause?	Error Locations	Mode	Conflicting Operations	Failure Symptoms	Num. of Proc.
emulate	04/2011	Yes	Yes	within an epoch	passive	get and load/store	incorrect result	2
BT-broadcast	06/2004	Yes	Yes	within an epoch	active	get and load	program hang	2
lockopts-rev	r10308	Yes	Yes	across processes	passive	put/get and load/store	incorrect result	64
pingpong-inj	3.0.3	Yes	Yes	across processes	passive	put and put	incorrect result	64
jacobi-inj	09/2008	Yes	Yes	across processes	active	put and get	incorrect result	64

specific program rules. In contrast, previous statistics-based approaches [26], [27] can detect only those bugs occurring in large scale because such approaches need to collect a large amount of statistical data at runtime.

To the best of our knowledge, MC-Checker is the first comprehensive approach to detect memory consistency errors in MPI one-sided applications. While Marmot [10] can detect some one-sided errors in MPI applications, it is limited to deadlock and parameter errors. A model-checking approach [28] was proposed to detect deadlocks in MPI one-sided applications; however, it cannot handle memory consistency errors. A recently proposed work, SyncChecker [29], can detect the errors occurring within an epoch; however, it cannot detect memory consistency errors across processes.

A. Case Studies

This section presents two representative real-world bug cases evaluated in our experiments. One is from the algorithm in the appendix of a paper by Leucke et al. [21], while the other one is from the RMA test case in the MPICH package.

1) Case 1: Conflicting `MPI_Get` and load operations in BT-broadcast: This memory consistency error was found in a binary tree broadcast algorithm [21]. The error causes the program to execute a while loop forever. In our evaluation, we implement the algorithm in C code and run it in the cluster. Figure 6 shows the buggy code extracted from the algorithm. Line 1 and line 8 form a one-sided communication epoch. Within the epoch, line 3 performs a store operation to initialize the local variable `check`, which is followed by a while loop between line 4 and line 6. The `MPI_Get` will update the value of `check` from a remote process. However, it may not be completed until the end of the epoch at line 8 because of the nonblocking nature of one-sided communication. As a result, the program will execute the while loop forever as the value of variable `check` is always 0.

This error can be triggered by running BT-broadcast in two processes. After being applied to this program, MC-Checker reports that a local load operation is conflicting with `MPI_Get`. In addition, MC-Checker identifies the locations of these two conflicting operations, which are line 4 and line 5 in Figure 6. With such detailed diagnostic information, programmers can easily locate the bug and fix it.

2) Case 2: Conflicting `MPI_Put/Get` and load/store operations in lockopts: This memory consistency error was found in the RMA test case in the MPICH package with svn revision number 10308. It can cause the program to yield nondeterministic results. Figure 7 illustrates the buggy scenario extracted from the source code. Because of space limits, we show only the code snippet of part of the buggy areas. The double directional arrows show the synchronization points with happens-before and consistency ordering relations. This is a revised version of the bug. The original bug has an exclusive lock in P0. We can see that the load and store operations in section A are within the concurrent region of the `MPI_Put` (abbreviated as `Put` in the figure) in section D. By checking with the compatibility table, we see that the load/store operation is conflicting with `MPI_Put`. Thus the program will generate incorrect results. For the original bug with the exclusive lock, we can also detect it but report only a warning. We need to rely on programmers to identify its buggy scenario from the warning.

```
1: MPI_Win_fence(0, win);
2: ...
3: check = 0;
4: while(check == 0){  // buggy load access
5:   MPI_Get(&check, 1, MPI_DOUBLE, ...);
6: }
7: ...
8: MPI_Win_fence(0, win);
```

Fig. 6: Bug case 1: Conflicting `MPI_Get` and load operations in BT-broadcast. Note that the load operation of `check` is conflicting starting from the second iteration of while loop.

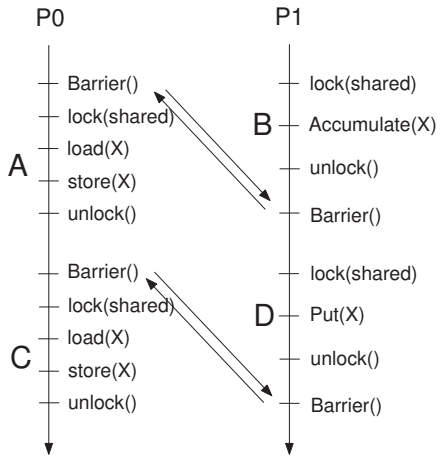


Fig. 7: Bug case 2: Conflicting operations in lockopts-rev. load/store in section A is concurrent with put in section D.

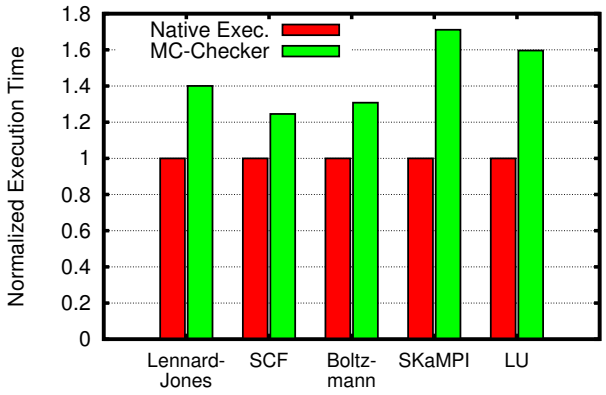


Fig. 8: Runtime overhead of MC-Checker. "Native" means execution without MC-Checker. "MC-Checker" means execution with MC-Checker's Profiler.

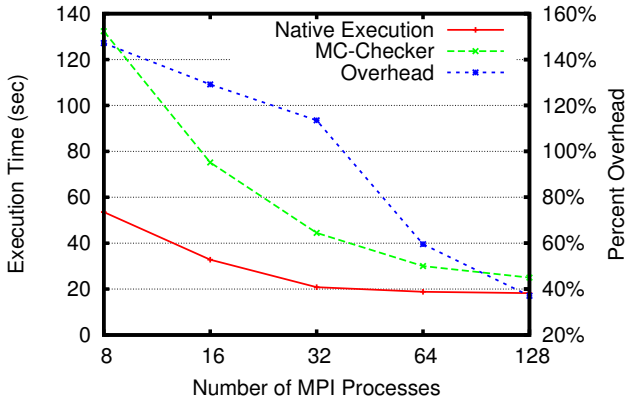


Fig. 9: Scalability study on benchmark LU

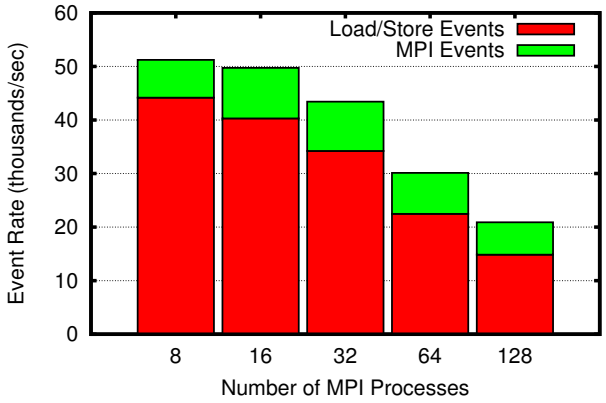


Fig. 10: Rate of profiling runtime events on benchmark LU

This bug can be triggered by running lockopts in 64 processes. After being applied to this bug case, MC-Checker reports the conflicting put and load/store operations. Additionally, MC-Checker pinpoints the location of the conflicting operations so that programmers can quickly identify the bug.

From this bug case, we can also see that one can easily make mistakes when writing MPI one-sided applications. The lockopts bug occurred in an RMA test case that is part of the MPICH MPI implementation, and was written by an MPI expert. Thus, tools to assist programmers in detecting bugs are valuable in aiding the development of one-sided applications.

B. Runtime Overhead

Figure 8 shows the execution time of five applications without and with MC-Checker's Profiler. The execution time for each application is normalized to the native execution, namely, the original program without MC-Checker. As shown in the figure, the runtime overhead incurred by MC-Checker's Profiler is low, ranging from 24.6% to 71.1%, with an average of 45.2%.

The reason for the low runtime overhead is that MC-Checker logs only the necessary load/store and MPI function-level events. This is the benefit from static analysis. Without static analysis, MC-Checker may cause hundreds of times more overhead because it needs to instrument all memory load/store accesses, as demonstrated in previous studies such as SyncChecker [29] and Purify [30]. Although SyncChecker's Profiler performs runtime optimizations, the average overhead is still 3.85 times.

Figure 9 shows the scalability study of MC-Checker's Profiler on the LU benchmark. As shown in the figure, the runtime overhead decreases from 147.2% to 37.1% when the number of processes increases from 8 to 128. The reason is that Profiler logs the runtime events into the local disk independently for each process. As shown in Figure 10, the rate of profiling runtime events, especially load/store events, decreases while the number of processes increases, which explains the reason that overhead drops.

The overhead of MC-Checker comes mainly from memory access instrumentation, which is proportional to the computation events in each node. Communication event instrumentation is light weight. The LU benchmark scalability experiment is strong scaling experiment. When the number of nodes increases, the rate of computation events decreases. Therefore the rate of profiling events decreases, leading to decreasing runtime overhead. For weak scaling experiments, the workload assigned to each processing node stays constant, we expect a constant overhead when the number of nodes increases.

This subsection focuses on the runtime overhead of Profiler, the online component of MC-Checker. While MC-Checker analyzes the traces offline, we can extend it to perform online analysis by leveraging streaming processing algorithms in the future.

VIII. Related Work

Our work is related to the following categories of studies.

Bug detection for MPI one-sided programs. Only a few studies have been done on detecting bugs in MPI one-sided programs. Marmot [10] focuses on checking parameter errors during MPI RMA calls and deadlock problems. Model checking [28] can also uncover potential deadlock problems. Scalasca [31] can identify potential performance bottlenecks in RMA programs and offer guidance in exploring their causes.

Our work, however, focuses on memory consistency errors in MPI RMA programs.

Bug detection and validation for general MPI programs. Tools for detecting MPI application bugs have been broadly investigated within the community. Tools and algorithms for deadlock detection generate wait-for graphs to detect cycles in blocking MPI calls [11], [12]. Validation tools have been developed to prove that MPI programs are deadlock-free [32], [33]. To our best knowledge, the work we present is the first to tackle memory consistency error detection for MPI one-sided programs.

Bug detection for other one-sided models. UPC-Thrille is a tool for data race detection in Unified Parallel C (UPC) programs [34], [35]. UPC is a partitioned global address space parallel programming model, which uses a one-sided communication runtime system. Important differences in the UPC and MPI memory models distinguish this work from ours. For but one example, UPC's memory model is coherent and relies on a global total ordering for strict accesses. In contrast, the MPI memory model is designed for noncoherent platforms and does not provide strict ordering guarantees.

Shared-memory data race detection. In shared-memory programs, much research has been conducted to detect memory consistency errors, also called data races. Some approaches [1], [2], [3] detect data races via static analysis, while others [4], [5], [6], [7] leverage dynamic methods that use algorithms such as lock-set to automatically detect data races in shared memory. Several key differences can be distinguished between shared-memory data race detection and detecting memory consistency errors in the MPI one-sided memory model. In the MPI RMA model, only the owning process can perform direct load and store accesses; all other accesses are performed through library calls, which are coarser in granularity and can be easily tracked through library interposition. In addition, in the MPI RMA model, only buffers that are involved in RMA calls can be locations for memory consistency errors, whereas any location is possible in shared-memory programs. Because of the above key differences, the bug detection algorithms are different.

IX. Conclusions

This paper presents MC-Checker, a new method to detect memory consistency errors in MPI one-sided applications. Based on collected runtime events, MC-Checker checks the MPI one-sided calls and local load/store operations against the compatibility tables to see whether they have memory consistency errors. If any error is found, MC-Checker reports the bug, along with diagnostic information.

We have built a prototype of MC-Checker on Linux⋆. Our evaluation with three real-world and two injected bugs in five MPI one-sided applications shows that MC-Checker is effective in detecting memory consistency errors. In addition, MC-Checker provides diagnostic information to help locate and fix the bugs. Furthermore, our experiments with the five applications show that MC-Checker incurs low runtime overhead, ranging from 24.6% to 71.1%, with an average of 45.2%. These results indicate that MC-Checker can be applied to production runs without degrading performance substantially.

X. Acknowledgments

We thank the anonymous reviewers for invaluable feedback. This work was supported in part by the U.S. NSF grants #CCF-0953759 (CAREER Award), #CCF-1218358, and #CCF-1319705, by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357, by an allocation of computing time from the Ohio Supercomputer Center, and by the CAS/SAFEA International Partnership Program for Creative Research Teams.

⋆Other names and brands may be claimed as the property of others.

References

[1] D. Engler and K. Ashcraft, "Racerx: effective, static detection of race conditions and deadlocks," in *SOSP*, 2003.

[2] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for Java," in *PLDI*, 2006.

[3] P. Pratikakis, J. S. Foster, and M. Hicks, "Locksmith: context-sensitive correlation analysis for race detection," in *PLDI*, 2006.

[4] M. D. Bond, K. E. Coons, and K. S. McKinley, "Pacer: proportional detection of data races," in *PLDI*, 2010.

[5] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan, "Efficient and precise datarace detection for multithreaded object-oriented programs," in *PLDI*, 2002.

[6] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, 1997.

[7] Y. Yu, T. Rodeheffer, and W. Chen, "Racetrack: efficient detection of data race conditions via adaptive tracking," in *SOSP*, 2005.

[8] Z. Chen, Q. Gao, W. Zhang, and F. Qin, "Flowchecker: Detecting bugs in mpi libraries via message flow checking," in *Supercomputing*, 2010.

[9] J. DeSouza, B. Kuhn, B. R. de Supinski, V. Samofalov, S. Zheltov, and S. Bratanov, "Automated, scalable debugging of mpi programs with Intel® Message Checker," in *SE-HPCS*, 2005.

[10] B. Krammer and M. M. Resch, "Correctness checking of MPI one-sided communication using Marmot," in *EuroPVM/MPI*, 2006.

[11] B. Krammera, K. Bidmona, M. S. Mullera, and M. M. Rescha, "MARMOT: An MPI analysis and checking tool," in *PARCO*, 2003.

[12] J. S. Vetter and B. R. de Supinski, "Dynamic software testing of MPI applications with Umpire," in *Supercomputing*, 2000.

[13] MPI Forum, "MPI: A message-passing interface standard version 2.2," University of Tennessee, Knoxville, Tech. Rep., 2009.

[14] ——, "MPI: A message-passing interface standard version 3.0," University of Tennessee, Knoxville, Tech. Rep., Sep. 2012.

[15] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, pp. 558–565, July 1978.

[16] E. Lusk, S. Pieper, and R. Butler, "More Scalability, Less Pain," *SciDAC Review*, no. 17, pp. 30–37, 2010. [Online]. Available: http://www.scidacreview.org/1002/html/adlb.html

[17] S. C. Pieper and R. B. Wiringa, "Quantum Monte Carlo calculations of light nuclei," *Annual Review of Nuclear and Particle Science*, vol. 51, no. 1, pp. 53–90, 2001. [Online]. Available: http://bit.ly/143fd6u

[18] Ewing Lusk, author of ADLB library, Personal correspondence, 2013, Argonne National Laboratory Emeritus & Argonne Distinguished Fellow.

[19] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004.

[20] Ohio Supercomputer Center, http://www.osc.edu.

[21] G. R. Luecke, S. Spanoyannis, and M. Kraeva, "The performance and scalability of shmem and mpi-2 one-sided routines on a SGI Origin 2000 and a Cray T3E-600: Performances," *Concurr. Comput. : Pract. Exper.*, vol. 16, no. 10, pp. 1037–1060, Aug. 2004.

[22] "MPICH2: A high-performance and widely portable implementation of the message passing interface (MPI) standard," http://www.mcs.anl.gov/research/projects/mpich2.

[23] J. Dinan, P. Balaji, J. R. Hammond, S. Krishnamoorthy, and V. Tipparaju, "Supporting the global arrays pgas model using mpi one-sided communication," in *IPDPS*, 2012.

[24] "Global Arrays," http://hpc.pnl.gov/globalarrays.

[25] "SKaMPI," http://liinwww.ira.uka.de/~skampi.

[26] Q. Gao, F. Qin, and D. K. Panda, "Dmtracker: finding bugs in large-scale parallel programs by detecting anomaly in data movements," in *Supercomputing*, 2007.

[27] A. V. Mirgorodskiy, N. Maruyama, and B. P. Miller, "Problem diagnosis in large-scale computing environments," in *Supercomputing*, 2006.

[28] S. Pervez, G. Gopalakrishnan, R. M. Kirby, R. Thakur, and W. Gropp, "Formal verification of programs that use MPI one-sided communication," in *EuroPVM/MPI*, 2006.

[29] Z. Chen, X. Li, J.-Y. Chen, H. Zhong, and F. Qin, "SyncChecker: Detecting synchronization errors between mpi applications and libraries," in *IPDPS*, 2012.

[30] R. Hastings and B. Joyce, "Purify: Fast detection of memory leaks and access errors," in *Winter USENIX Conference*, 1992.

[31] M.-A. Hermanns, M. Miklosch, D. Böhme, and F. Wolf, "Understanding the formation of wait states in applications with one-sided communication," in *EuroMPI*, 2013.

[32] S. Vakkalanka, M. Delisi, G. Gopalakrishnan, R. M. Kirby, R. Thakur, and W. Gropp, "Implementing efficient dynamic formal verification methods for MPI programs," in *EuroPVM/MPI*, 2008.

[33] A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, and R. Thakur, "Formal verification of practical MPI programs," in *PPoPP*, 2009.

[34] C.-S. Park, K. Sen, P. Hargrove, and C. Iancu, "Efficient data race detection for distributed memory parallel programs," in *Proc. 2011 Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11, 2011.

[35] C. S. Park, K. Sen, and C. Iancu, "Scaling data race detection for partitioned global address space programs," in *Proc. 27th Intl. ACM Conf. on Supercomputing*, ser. ICS '13, 2013.