LLNL-CONF-870387

VerifyIO: Ensuring Correctness of Consistency Semantics in Parallel I/O

C. Wang, Z. Zhu, K. Mohror, S. Neuwirth, M. Snir

October 9, 2024

39th IEEE International Parallel & Distributed Processing Symposium
Milan, Italy
June 3, 2025 through June 7, 2025

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

VerifyIO: Verifying Adherence to Parallel I/O Consistency Semantics

Chen Wang
Lawrence Livermore National Laboratory
Livermore, USA
0000-0001-9297-0415

Zhaobin Zhu
Johannes Gutenberg University Mainz
Mainz, Germany
0009-0005-6916-100X

Kathryn Mohror
Lawrence Livermore National Laboratory
Livermore, USA
0000-0002-1366-1655

Sarah Neuwirth
Johannes Gutenberg University Mainz
Mainz, Germany
0000-0001-7409-153X

Marc Snir
University of Illinois Urbana-Champaign
Champaign, USA
0000-0002-3504-2468

Abstract—**High-performance computing (HPC) applications generate and consume substantial amounts of data, typically managed by parallel file systems. These applications access file systems either through the POSIX interface or by using high-level I/O libraries. While the POSIX consistency model remains dominant in HPC, emerging file systems and popular I/O libraries increasingly adopt alternative consistency models that relax semantics in various ways, creating significant challenges for correctness and portability. This paper addresses these challenges by proposing a trace-driven I/O consistency verification workflow, implemented in our open-source tool, VerifyIO, which collects execution traces, detects data conflicts, and verifies proper synchronization against specified consistency models. Our extensive evaluation of 91 test case executions across three widely used I/O libraries with four I/O consistency models reveals critical consistency issues at both application and implementation levels.**

Index Terms—**Consistency Semantics, POSIX Semantics, MPI-IO, I/O Consistency, Parallel File System**

I. Introduction

HPC applications, such as scientific simulations and AI training, generate and process vast amounts of data, imposing significant demands on I/O systems to ensure efficient and correct data handling. These I/O demands are typically supported by parallel file systems. To ensure portability and compatibility, most widely deployed parallel file systems, such as Lustre [1] and GPFS [2], conform to the POSIX [3] standard, providing both the POSIX interface and its associated consistency semantics. HPC applications access these parallel file systems either directly through POSIX APIs or via higher-level I/O libraries (e.g., PnetCDF [4] and HDF5 [5]). These libraries offer more user-friendly interfaces and implement various parallel I/O optimizations such as I/O aggregation, data sieving, and asynchronous I/O. While these libraries may eventually invoke POSIX APIs for portability, they often adopt relaxed consistency models (weaker than POSIX) for better performance. For instance, HDF5 and PnetCDF, both built on MPI-IO [6], use MPI-IO's relaxed consistency model, which diverges from the strict guarantees of POSIX.

On the storage side, though many HPC systems continue to use POSIX-compliant file systems, recent trends show that emerging file systems such as BurstFS [7], UnifyFS [8], and GfarmBB [9]—are choosing to relax POSIX consistency semantics while maintaining the POSIX interface. By keeping the POSIX interface, they support legacy applications while relaxing consistency semantics to improve performance. However, this trade-off introduces new risks related to portability and correctness, especially when applications assume stricter POSIX guarantees.

A key question emerges: How can we ensure that applications behave correctly on systems with different consistency models? Even if an application and its associated I/O libraries are programmed using POSIX APIs, how can we verify that they adhere to the specific consistency rules of the underlying system—especially when those rules deviate from POSIX? Furthermore, if a program violates these semantics, how can we diagnose the cause of a violation, e.g., whether it is caused by the application or the underlying I/O library? Answering these questions is crucial for ensuring both the portability and correctness of HPC applications, and also help pave the road for adopting future non-POSIX systems. To the best of our knowledge, no existing study directly addresses these questions.

There are several challenges in tackling these questions. First, verifying correctness requires formal specifications of the target consistency model, as well as a rigorously designed verification algorithm. Second, different I/O systems may relax POSIX semantics in different ways, making it difficult to design a generic verification algorithm that can account for all possible variations. Third, as the I/O software stack deepens, HPC applications may involve multiple libraries and middleware, making it difficult to trace the root cause of semantics violations. Lastly, the solution must be insightful, easy-to-use, and capable of helping both application developers and I/O system designers identify and resolve consistency issues. This requires significant designing and engineering efforts.

To tackle these challenges, we propose a trace-driven four-step verification workflow that systematically collects execution traces and verifies their adherence to specific consistency models. We use the framework from [10] to specify I/O consistency models in a unified way. This aids in the design of a generic verification algorithm that can handle different consistency models. Further, we extend the framework to define the concept of a *properly-synchronized execution*, where no data conflicts occur, or all conflicts are properly synchronized according to the specified consistency model.

For any execution, our algorithm examines the traces to verify whether it is properly synchronized, i.e., whether it follows the rules of the specified consistency model. Although a properly synchronized execution does not guarantee that the entire application is synchronized correctly, because applications may follow different I/O execution paths. In our experience, this is rare though, most HPC applications tend to have a few if not one I/O path. More importantly, when an execution is found to be improperly synchronized, it indicates the presence of data races, suggesting potential consistency issues or implementation bugs in either the application or the I/O library.

We present VerifyIO, an open-source project that implements the proposed verification workflow. VerifyIO contains four components, one for each step of the workflow: (1) *a tracing tool* that collects execution traces with sufficient information for later steps, (2) *a conflict detection tool* that identifies data conflicts within the execution traces, (3) *a MPI matching tool* that matches MPI calls to establish the temporal order between all conflicting operations, and (4) *a verification tool* that checks whether identified conflicts are properly synchronized according to the target consistency model. Additionally, VerifyIO reports call-chain information for improperly synchronized conflicts, offering insights into whether the application or its I/O libraries are responsible for the issue. This information helps pinpoint the root cause of data races and can even expose underlying consistency bugs.

Overall, this paper makes the following key contributions:

- We present a trace-driven workflow for verifying I/O consistency semantics in HPC applications, implemented in the VerifyIO project. VerifyIO supports common I/O consistency models and reports detailed call-chain information to aid in diagnosing semantics violations.
- We develop a tracing library within VerifyIO that captures all necessary information for verification, supporting a complete set of APIs for three popular I/O libraries—a feature existing tracing tools do not offer.
- We conduct an extensive study using 91 test case executions from three I/O libraries, verifying each against four I/O consistency models. Our results show that six tests are not properly synchronized even under POSIX, and 28 exhibit synchronization issues across all four models. We perform an in-depth analysis of the detected data races, identifying several application- and implementation-level consistency issues that could lead to incorrect result.

II. Background

Before we dive into the proposed workflow and implementation, we use this section to give the background information on consistency models and I/O libraries, with a special focus on the ones that we will use in our evaluation.

A. Consistency Models

A consistency model defines the contract between the programmer and the system, specifying the rules under which shared data remains consistent. Adherence to the model ensures that the outcomes of read, write, and update operations are predictable and correct. While POSIX consistency is the dominant model in HPC, other consistency models are also used in real-world I/O libraries and file systems. As mentioned earlier, despite their differences, these models can be specified uniformly [10], which makes designing a generic verification algorithm feasible.

1) POSIX Consistency: The POSIX standard [3] defines a strong consistency model that requires all writes to be immediately visible to all subsequent reads. While this model is simple to maintain in single-node environments, it is expensive to implement at scale [11], [12]. Nevertheless, major parallel file systems such as Lustre [1], GPFS [2], and BeeGFS [13] continue to support POSIX consistency due to its compatibility and widespread adoption.

2) Commit Consistency: Commit consistency offers a relaxed model often used in user-level parallel file systems, such as BSCFS [14], UnifyFS [8], and SymphonyFS [15]. Here, synchronization is explicitly performed by issuing "commit" operations, typically by writers, to ensure data becomes globally visible. The data written prior to a commit is only visible after the commit operation completes. In practice, file systems using commit consistency may map a commit to an existing POSIX call; for example, UnifyFS uses the `fsync` call to signal a commit.

3) Session Consistency: Session consistency, also known as close-to-open consistency, is another relaxed model that synchronizes data between processes when one process closes a file and another subsequently opens it. This model addresses cases where global visibility (ensured by commit consistency) is unnecessary, such as when only a subset of processes perform reads. Session consistency uses `close` and `open` operations to control visibility between processes.

4) MPI-IO Consistency: MPI-IO [6], a part of the MPI standard [16], specifies MPI's I/O functionalities. MPI-IO's relaxed model ensures sequential consistency for conflicting accesses through a *sync-barrier-sync* construct. In this pattern, `MPI_File_open`, `MPI_File_close`, and `MPI_File_sync` serve as synchronization points for flushing or retrieving data, while barriers (e.g., `MPI_Barrier`, or point-to-point communications like `MPI_Send`/`MPI_Recv`) ensure proper ordering.

B. I/O Libraries

I/O libraries provide portability, efficiency, and scalability at different layers of the I/O stack. This work focuses on HDF5 [5], NetCDF [17], and PnetCDF [4], which are widely used for managing large, multi-dimensional datasets in HPC. These libraries are designed to provide machine-independent data formats and high-level APIs for accessing scientific data. Ensuring their correctness, especially with respect to data consistency, is critical given their extensive use in real-world applications.

HDF5 [5] is a data model and library designed for the storage and management of complex data objects. It provides advanced data structures, such as groups and datasets, to organize and store scientific data. HDF5 supports various optimizations like hyperslab selections for non-continuous access and asynchronous I/O, allowing applications to overlap I/O with computation to improve performance.

NetCDF [17] is designed for array-oriented scientific data, with a simple data model consisting of dimensions, variables, and attributes. It allows users to define and store multi-dimensional arrays, and is commonly used in climatology, meteorology and oceanography application. The latest version of NetCDF can use either HDF5 or PnetCDF as its backend.

PnetCDF (Parallel netCDF) [4] is a parallel I/O library tailored for high-performance applications that use the NetCDF format. Built on MPI-IO, PnetCDF offers over 900 APIs for handling files, variables, and attributes, with features like collective I/O and non-blocking operations to maximize performance in parallel environments.

III. Methodology

The proposed workflow is built around a generic semantics verification algorithm designed for commonly used I/O models in HPC systems. These models are specified using a unified framework proposed in [10]. We adopt the same terminology and define when an execution is considered properly-synchronized for a given consistency model, i.e., the execution abides the rules of the consistency model. Based on this definition, we then outline the overall verification workflow.

A. A Formal Definition

We define two types of I/O operations: An I/O operation is either a *data operation* or a *synchronization operation*. Data operations are operations that read or write storage, such as `fread` or `fwrite`. Data operations include the specification of the storage location to be read or written. Synchronization operations are special I/O operations that can be used to impose order on data operations, such as `fsync`, `fopen`, or `fclose`. Synchronization operations are model-specific.

We consider the execution of a multiprocess program, in an environment that provides well-defined mechanisms to synchronize concurrent processes, such as MPI message-passing. These mechanisms define a *program order* and *synchronization order* on the executed operations of the program:

Def. 1 [Program Order ($\xrightarrow{po}$)]: The program order of a process is a total order on the execution of the process' operations as specified by the program text. To keep the discussion simple, we ignore the extensions needed to deal with multithreaded processes.

Def. 2 [Synchronization Order ($\xrightarrow{so}$)]: A synchronization order is a partial order specified between operations executed by distinct processes. This partial order is consistent with the program order, and $\xrightarrow{po} \cup \xrightarrow{so}$ is acyclic.

A properly-synchronized execution is defined as follows.

Def. 3 [Happens-Before Order ($\xrightarrow{hb}$)]: The happens-before order of an execution is the transitive closure of $\xrightarrow{po}$ $\cup \xrightarrow{so}$. The outcome of a parallel execution should be as if all instructions were executed in the order specified by $\xrightarrow{hb}$. Thus, if ow and or are, respectively, a write and a read to the same location, and $ow \xrightarrow{hb} or$, then or will return the value written by ow, unless there is another store ow' to the same location such that $ow \xrightarrow{hb} ow' \xrightarrow{hb} or$.

Def. 4 [Conflict]: Two data operations *conflict* iff (if and only if) their access ranges (contiguous file offset ranges) overlap, and at least one of them is a write.

Def. 5 [Minimum Synchronization Construct (*MSC*)]: An *MSC* specifies a minimum sequence of synchronization operations required to synchronize two conflicting data operations. An *MSC* consists of k synchronization operations and $k+1$ edges, where $k \geq 0$:

$$MSC = \xrightarrow{r_0} S_1 \xrightarrow{r_1} S_2 \xrightarrow{r_2} \ldots \xrightarrow{r_{k-1}} S_k \xrightarrow{r_k}$$

For each i, $1 \leq i \leq k$ and $S_i \in S$, where S is the set of synchronization operations to be defined by the specific consistency model. For each j, $0 \leq j \leq k$ and $\xrightarrow{r_j} \in \{\xrightarrow{po}, \xrightarrow{hb}\}$.

Table I shows the S and *MSC* for the four consistency models discussed in Sec II-A.

Def. 6 [Properly-Synchronized Relation ($\xrightarrow{ps}$)]: Two conflicting data operations X and Y are properly synchronized, i.e., $X \xrightarrow{ps} Y$, iff one of the following holds:

1) X is a read operation and $X \xrightarrow{hb} Y$.
2) X is a write operation, and there exists an MSC between X and Y in the happens-before order.

Def. 7 [Data Race]: Two data operations X and Y in an execution form a *data race* iff they conflict and they are not properly synchronized.

Def. 8 [Properly-Synchronized Execution]: An execution is properly synchronized iff for every sequentially consistent execution of the program, all I/O operations can be distinguished by the system as either data or synchronization, and there are no data races in the execution.

B. Workflow and Challenges

The proposed workflow aims to verify whether a given execution is properly synchronized under a specific consistency model. It is structured into four steps, as illustrated in Fig. 1. Below, we outline each step and discuss its associated challenges.

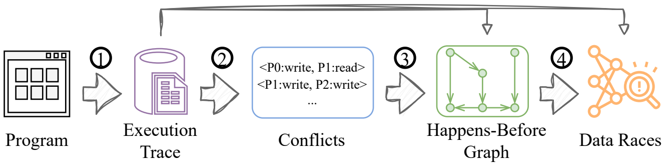


Fig. 1: Verification workflow

Consistency Models	***S***	***MSC***
POSIX Consistency	$\{\}$	$\xrightarrow{hb}$
Commit Consistency	$\{$`commit`$\}$	$\xrightarrow{hb}$ `commit` $\xrightarrow{hb}$
Session Consistency	$\{$`session_close, session_open`$\}$	$\xrightarrow{po}$ `session_close` $\xrightarrow{hb}$ `session_open` $\xrightarrow{po}$
MPI-IO Consistency	$\{$`MPI_File_sync,` `MPI_File_close,` `MPI_File_open`$\}$	$\xrightarrow{po} s_1 \xrightarrow{hb} s_2 \xrightarrow{po}$ $s_1 \in \{$`MPI_File_close, MPI_File_sync`$\}$ $s_2 \in \{$`MPI_File_sync, MPI_File_open`$\}$

TABLE I: The synchronization operation set (*S*) and the minimum synchronization construct (*MSC*) for four commonly-seen storage consistency models.

1) Generating Execution Trace: The first step involves running the target program and collecting an execution trace. This trace is used for later verification steps. The main challenge here is ensuring that the tracing tool collects all the necessary function calls and their arguments, including the complete call chain for any potential violations. The trace needs to cover multiple I/O layers (e.g., NetCDF, PnetCDF, MPI-IO, and POSIX) to capture all relevant function calls. Further, the desired tracing tool must also handle additional complexities, such as maintaining mappings between file descriptors and file names, as well as distinguishing between nested open/close calls on the same file. While many tracing tools exist, none fully meet our requirements.

2) Detecting Conflicts: Once the execution trace is generated, the next step is to detect conflicts (Def. 4) between I/O operations. Conflicts occur when two or more operations access the same file location, and at least one is a write. Challenges arise when retrieving access details for operations that do not explicitly include file locations in their arguments (e.g., `fwrite`). In such cases, previous metadata calls like `lseek` or `fseek` must be examined to recover the location information. Additionally, the detection algorithm must handle corner cases, such as I/O operations using different file handles for the same file. The output of this step is a list of *conflict groups*. Each group contains a conflicting operation and a mapping of ranks (processes) to other conflicting operations. We denote a conflict group as (X, ζ), where X is a data operation, ζ is the mapping, then for each $Y \in \zeta_r$, X and Y form a *conflict pair*.

3) Establishing Happens-Before Order: For each conflict, the happens-before order (Def. 3) must be established to determine whether the conflicting operations are properly synchronized. This order is derived from the program order and the synchronization order defined by the MPI calls. Thus, our tracing library also needs to intercept and store MPI calls. This step matches the captured MPI calls (essentially replaying them) to establish the synchronization order. To do so, we need to collect many extra information. For example, to match `MPI_Send` to a `MPI_Irecv`, we need to intercept the `MPI_Wait` or `MPI_Test` on the receive side. Additionally, we need to keep track of the request id of each MPI request.

4) Verifying Consistency Semantics: In the final step, the verification algorithm examines these conflicts to determine whether they are properly synchronized (Def. 6) according to the target consistency model. One challenge in this step is the potentially large number of conflict pairs to verify. In some cases, even small programs can produce millions of conflicts (e.g., *pmulti_dset* in Section V), making it essential to optimize the verification process for efficiency. In particular, pruning strategies are necessary to reduce the number of conflicts to examine. Finally, when a data race (Def. 7) is detected, the algorithm should report the complete call chain involved, helping users pinpoint the root cause.

IV. VerifyIO

VerifyIO is an open-source project[1] that implements the entire workflow and addresses the challenges outlined in Section III-B. Its primary goal is to provide the I/O community with a tool for verifying and ensuring the correctness of I/O semantics, especially as we move away from POSIX. We aim to make VerifyIO as user-friendly, generic, and flexible as possible. Specifically, the tracing library (step one) is designed for easy extension to support additional I/O libraries, while the verification algorithm (step four) is capable of working with various consistency models.

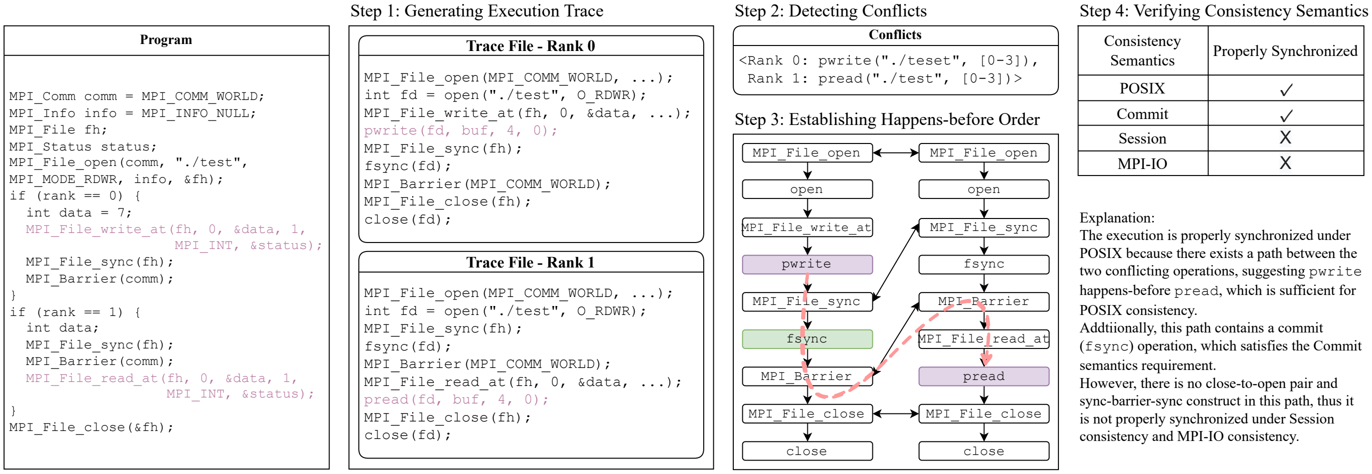
This section follows the four steps of the proposed workflow described earlier, with each subsection detailing the implementation of one workflow step. An illustrative example is provided in Fig. 2.

A. Generating Execution Trace

The tracing tool is a crucial component of VerifyIO, as it collects the data necessary for subsequent steps. In particular, it captures I/O calls for conflict detection, MPI calls for establishing synchronization order, and both I/O and MPI calls for the final semantics verification and call-chain reporting.

While many tracing tools exist, none meet all the requirements of VerifyIO as discussed in Section III-B. Among the tools we studied, Recorder [18] came closest to fulfilling our needs. Recorder intercepts POSIX, MPI, MPI-IO, and HDF5 functions, capturing all arguments of each intercepted call. However, it only supports a single I/O library (HDF5) and provides incomplete coverage—supporting 84 HDF5 functions, while the latest HDF5 version includes over 700 APIs.

[1] https://https://github.com/wangvsa/VerifyIO



Program

```
MPI_Comm comm = MPI_COMM_WORLD;
MPI_Info info = MPI_INFO_NULL;
MPI_File fh;
MPI_Status status;
MPI_File_open(comm, "./test",
MPI_MODE_RDWR, info, &fh);
if (rank == 0) {
  int data = 7;
  MPI_File_write_at(fh, 0, &data, 1,
                    MPI_INT, &status);
  MPI_File_sync(fh);
  MPI_Barrier(comm);
}
if (rank == 1) {
  int data;
  MPI_File_sync(fh);
  MPI_Barrier(comm);
  MPI_File_read_at(fh, 0, &data, 1,
                   MPI_INT, &status);
}
MPI_File_close(&fh);
```

Step 1: Generating Execution Trace

Trace File - Rank 0

```
MPI_File_open(MPI_COMM_WORLD, ...);
int fd = open("./test", O_RDWR);
MPI_File_write_at(fh, 0, &data, ...);
pwrite(fd, buf, 4, 0);
MPI_File_sync(fh);
fsync(fd);
MPI_Barrier(MPI_COMM_WORLD);
MPI_File_close(fh);
close(fd);
```

Trace File - Rank 1

```
MPI_File_open(MPI_COMM_WORLD, ...);
int fd = open("./test", O_RDWR);
MPI_File_sync(fh);
fsync(fd);
MPI_Barrier(MPI_COMM_WORLD);
MPI_File_read_at(fh, 0, &data, ...);
pread(fd, buf, 4, 0);
MPI_File_close(fh);
close(fd);
```

Step 2: Detecting Conflicts

Conflicts

```
<Rank 0: pwrite("./teset", [0-3]),
 Rank 1: pread("./test", [0-3])>
```

Step 3: Establishing Happens-before Order

Step 4: Verifying Consistency Semantics

Consistency Semantics	Properly Synchronized
POSIX	✓
Commit	✓
Session	X
MPI-IO	X

Explanation:
The execution is properly synchronized under POSIX because there exists a path between the two conflicting operations, suggesting `pwrite` happens-before `pread`, which is sufficient for POSIX consistency.
Addtiionally, this path contains a commit (`fsync`) operation, which satisfies the Commit semantics requirement.
However, there is no close-to-open pair and sync-barrier-sync construct in this path, thus it is not properly synchronized under Session consistency and MPI-IO consistency.

Fig. 2: And example of the verification workflow. The target program performs a write on rank 0 followed by a read on rank 1, with I/O operations carried out via MPI-IO. In the first step, the program is executed, and an execution trace is collected. The trace captures not only the MPI-IO calls made by the program but also the underlying POSIX calls invoked by the MPI-IO implementation. In the second step, conflicts in the trace are identified. Here, the `pwrite` operation conflicts with the `pread` operation, as both attempt to access the first four bytes of the same file. Next, the happens-before relationship is established using program order and synchronization order. In the resulting happens-before graph, each node represents a function call, while each edge denotes a happens-before relationship. In the final step, the workflow verifies whether all detected conflicts are properly synchronized based on the specified consistency semantics. In this example, `pwrite` and `pread` are properly synchronized under the POSIX and Commit consistency models (indicated by the pink path), but they are not properly synchronized under the Session or MPI-IO semantics.

To address these gaps, we extended Recorder, creating a new version called Recorder$^+$, which can be easily adapted to support additional functions and libraries. Recorder$^+$ intercepts I/O calls through LD_PRELOAD and redirects them to the corresponding wrapper functions. The wrapper function records pre-invocation information (e.g., entry timestamp and thread id), invokes the original function, stores all runtime arguments, then returns to the original caller. In contrast to the original Recorder, where these wrappers were manually written (making it difficult and error-prone to extend), Recorder$^+$ uses a redesigned and automated approach, as shown below.

```
wrapper(func, ret_type, n_args, args) {
    prologue();
    ret_type ret = func(args);
    epilogue(n_args, args);
    return ret;
}
```

The `prologue` and `epilogue` handle the storage of pre- and post-invocation arguments and are implemented using macros to support arbitrary argument types. The wrapper function is also implemented as a macro, removing dependencies on function signatures. In companion, we developed a code-generation tool that takes a *function signature file* (think it as a header file) as input and automatically generates wrapper functions for each function in the file. The generated code can be easily compiled as a plugin to extend Recorder$^+$ to support new functions. In this work, Recorder$^+$ covers the complete function sets of three I/O libraries, as shown in Table II.

The remaining components of Recorder, such as compression and caching, are remain unchanged. Recorder typically incurs less than 10% of execution overhead to collect all the required information [18]. We observed similarly results for Recorder$^+$. Although the tracing tool's overhead is not a primary concern for this work, we include this information for completeness.

Tracing Tool	HDF5	NetCDF	PnetCDF
Recorder	84	-	-
Recorder$^+$	749	300	915

TABLE II: Supported functions comparison. Recorder$^+$ provides full coverage for HDF5, NetCDF, and PnetCDF APIs.

B. *Detecting Conflicts*

The second step detects conflicting operations in the execution trace. We first encode each data operation as an interval tuple $I = (fid, type, os, oe)$, where fid is a unique file identifier, $type$ is a binary indicator for read or write, and os/oe are the access start/end offsets. The conflict detection algorithm scans the trace to retrieve interval lists for each file and reports conflicts according to Def. 4.

A unique file identifier is necessary because the same file can be accessed using different interfaces with varying file handle representations. For example, it is possible to write to the same file using `pwrite` and `fwrite` simultaneously, as shown below, even though they use different file handle types (`int` vs `FILE*`). To address this, we track each file's open call in the execution trace and assign it a unique identifier, associating subsequent I/O calls with this identifier.

```
pwrite(int fd, void *buf, size_t count, off_t off);
fwrite(void* buf, size_t size, size_t count,
                              FILE* stream);
```

Access sizes can be directly retrieved from the trace (e.g., from the third argument of `pwrite` and by multiplying the second and third arguments for `fwrite`). However, determining access locations is not always straightforward. Functions like `fwrite` lack explicit offset arguments, so we derive them from earlier metadata operations. We maintain a (FP, EOF) pair for each opened file, where FP is the current file pointer (offset) and EOF is the current file size. Whenever we encounter an I/O operation, we update the (FP, EOF) pair accordingly. For example, a `lseek(fd, 0, SEEK_END)` call sets $FP = EOF$, and a `write(fd, buf, 100)` call moves FP 100 bytes forward and set $EOF = FP$ if $FP > EOF$. Using the update-to-date (FP, EOF), we can accurately track offsets for each I/O operation.

The conflict detection algorithm proceeds by marking conflicts between overlapping intervals where at least one is a write, as shown below:

```
// IS: intervals groupd by the file identifier
// and sorted by the start offset
void conflict_detection(Interval* IS) {
  for each fid:
    M = len(IS[fid])
    for i = 0 .. M:
      for j = i .. M:
        I = IS[fid][i]; J = IS[fid][j];
        if (J.os > I.oe)
          // subsequent Intervals (Js) will
          // not conflict with I
          break;
        if (I.type == WRITE || J.type == WRITE)
          conflicts[i, j] = TRUE;
}
```

C. Establishing Happens-before Order

In this step, we match all MPI calls recorded in the execution trace to establish synchronization orders between I/O operations. MPI communications are categorized into two types: point-to-point and collective. Point-to-point calls involve two processes, while collective calls involve multiple processes within the same communicator. The synchronization imposed by these MPI calls form the happens-before order among other I/O operations. For instance, in Fig. 2, the `MPI_Barrier` calls impose an order between the `pwrite` operation from rank 0 and the `pread` operation from rank 1.

Point-to-point calls are matched by comparing the destination, source, and tag arguments. For calls with wildcard arguments (i.e., `MPI_ANY_TAG` and `MPI_ANY_SRC`), we recover and compare the actual tag and source information from the `MPI_Status` argument returned by the receiver.

Collective calls are matched based on their communicator and in program order, e.g., two `MPI_Barrier` calls on the same communicator will never be matched out of order. One complexity arises with user-created communicators (e.g., not the builtin ones like `MPI_COMM_WORLD`). To handle this, we track communicator creation and duplication operations at tracing time and assign each communicator a globally unique identifier. When matching collective calls, we simply compare the communicator identifier passed by each process.

Handling non-blocking calls, such as `MPI_Isend` and `MPI_Iallreduce`, is especially tricky. These calls must be matched with the corresponding `MPI_Wait*` or `MPI_Test*` calls. It can be particularly difficult when they are paired with `MPI_Testsome` calls, since the associated `MPI_Request` may not yet be complete. We must first determine which requests have completed, then retrieve and compare them against the request generated by the original non-blocking call. While we cannot cover all corner cases due to space constraints, our implementation handles the most common scenarios, and we manually confirmed that it correctly matches all MPI calls used in our evaluations.

Once all MPI calls are matched, we construct a *happens-before graph*, denoted as $G = (V, E)$. Here, V represents the vertices of G, each corresponding to either a conflicting I/O operation or a synchronization event. The edges, E, capture the happens-before relationships between operations, derived from program order and synchronization order. For two vertices v_1 and v_2 in G, $v_1 \xrightarrow{hb} v_2$ if there is a path from v_1 to v_2.

D. Verifying Consistency Semantics

In the final step, VerifyIO examines the detected conflicts and determines whether they are properly synchronized. If a data race is found, the algorithm reports the entire call-chain to help identify the root cause.

A naive implementation would exhaustively check each conflicting pair, which can be expensive in the case of large numbers of conflicts. To optimize this, VerifyIO applies runtime pruning techniques to reduce unnecessary checks. As described earlier, we use *conflict groups* to organize conflicts based on process ranks and program order. The algorithm iterates over each conflict group, with four runtime pruning opportunities, as illustrated in Fig. 3. Here, X represents a data operation invoked by process P_a, and $Y_1, Y_2, \ldots, Y_n$ represent conflicting data operations (sorted by program order) invoked by process P_b. The four pruning scenarios are:

1) If $X \xrightarrow{ps} Y_1$, then $X \xrightarrow{ps} Y_i$ for all $i \in [2, n]$.
2) Conversely, if $Y_n \xrightarrow{ps} X$, then $Y_i \xrightarrow{ps} X$ for all $i \in [1, n-1]$.
3) If X does not $\xrightarrow{ps} Y_n$, then it does not $\xrightarrow{ps} Y_i$ for any earlier Y_i, because if it did, it would also have to $\xrightarrow{ps} Y_n$.
4) Similarly, if Y_n does not $\xrightarrow{ps} X$, then none of the earlier Y_i will $\xrightarrow{ps} X$.

Each pruning opportunity reduces the number of checks from n to 1. In practice, most conflict groups fall into one of these scenarios, meaning verification needs to be performed only once per conflict group. If a conflict group does not fit these scenarios, we fall back to checking every conflict pair within the group.

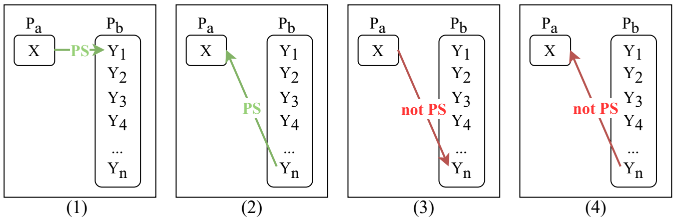


Fig. 3: Four runtime pruning scenarios, each reducing the number of check from n to 1.

As defined in Def. 6, the pair-wise verification of proper synchronization involves checking the existence of a happens-before order and verifying the MSC (Def. 5) between two I/O operations. MSC is determined by examining trace records, while the happens-before order can be determined using one of four approaches:

1) Vector Clock: Vector clock tracks event order across distributed processes and helps determine the partial order between two events. In our case, each event is a vertex in the happens-before graph. Since we already constructed the graph, computing vector clocks is straightforward. We perform a topological sort on the graph and propagate vector clocks through nodes in that order. This takes $\mathcal{O}(V + E)$ time, and once the clocks are computed, determining the happens-before order between any two operations takes $\mathcal{O}(1)$ time.

2) Graph Reachability: Graph reachability queries whether there exists a path from vertex v_1 to vertex v_2 in a directed acyclic graph. To determine if $X \xrightarrow{hb} Y$, we treat X and Y as vertices in the happens-before graph and use reachability algorithms. VerifyIO employs the reachability method from the NetworkX [19] package, which operates with a time complexity of $\mathcal{O}(V + E)$ per query.

3) Transitive Closure: Transitive closure captures all pairs of vertices (v_1, v_2) in a directed acyclic graph such that a path exists from v_1 to v_2. Computing the transitive closure takes $\mathcal{O}(V^3)$ in the worst case, but once done, it allows pair-wise verification in $\mathcal{O}(1)$ time.

4) On-the-fly: This algorithm determines the happens-before order at the time of verification, without the need to pre-build the happens-before graph. It matches MPI communications during each verification step to establish the order, checking for matching MPI events occurred in between the two conflicting I/O operations. In the worst case, when no match is found, the algorithm must go through all MPI calls of the involved processes.

After analyzing the complexity of each algorithm, it's clear that transitive closure and graph reachability are generally slower than vector clocks. Depending on the number of conflicts and the size of the graph, the on-the-fly algorithm may prove beneficial as it does not require pre-building the happens-before graph. Nevertheless, performance evaluation is not the primary focus of this work, and we leave it for future work. For validation purposes, we implemented all four approaches, using at least two in our experiments to ensure consistent results.

V. Evaluation

We selected 91 built-in tests from three widely-used I/O libraries as our target applications. These tests were chosen because they are self-contained and typically focus only on I/O operations. Additionally, these tests are usually created by the library developers to validate implementation correctness and system compatibility. As a result, they are more likely to adhere strictly to system rules and are less prone to library-usage errors.

We used the latest stable versions of the libraries available at the time of writing: HDF5 1.14.4.3, PnetCDF 1.13.0, and NetCDF 4.9.2. Our test cases include: 15 from HDF5, 17 from NetCDF, and 59 from PnetCDF. We verified each test against four I/O consistency models: POSIX, Commit, Session, and MPI-IO. All the tests were parallel in nature, and we executed them using the test scripts provided by the respective libraries. Initially, we ran the tests without VerifyIO to ensure they passed on our system, before rerunning them with VerifyIO to collect execution traces and verify their consistency semantics.

All experiments were conducted on Lassen at Lawrence Livermore National Laboratory, a system with 795 nodes, each equipped with an IBM Power9 CPU and 256 GB of memory. The I/O libraries were compiled using GCC and Spectrum MPI. The verification process was performed on a single node, with most runs completing within minutes, while a few larger tests took under an hour.

A. Summary

Fig. 4 illustrates the number of data races detected for each test across the four consistency models. Each row represents a test execution, and the columns display the number of data races detected for the POSIX, Commit, Session, and MPI-IO models. HDF5 tests are the largest among the three libraries, with many containing thousands of lines of code. In our experiments, over 800 million conflicts were detected across 7 of the 15 HDF5 test executions. In some tests, such as *shapesame* and *testphdf5*, hundreds of thousands of data races were reported under the weaker consistency models. In contrast, NetCDF tests are fewer in number and generally smaller in size. VerifyIO identified 53,793 conflicts across 9 of the 17 test executions, with over 9,000 data races detected under the relaxed models. For PnetCDF, VerifyIO identified a total of 107,185 conflicts across 12 out of 59 test executions, with 35,048 data races found under the relaxed consistency models. Additionally, three tests (marked in gray) were unable to complete the verification process due to unmatched MPI calls, which we will further discuss in Section V-D.

Table III summarizes the number of test executions that were not properly synchronized. Interestingly, we found that 6 out of 91 tests from all three libraries were not properly synchronized under POSIX semantics. This was unexpected, as data races under POSIX suggest that some read or write operations return undefined results. Notably, these 6 tests do not validate data integrity after each I/O operation, since they all pass on GPFS, a POSIX-compliant system. Section V-B will investigate the underlying causes.

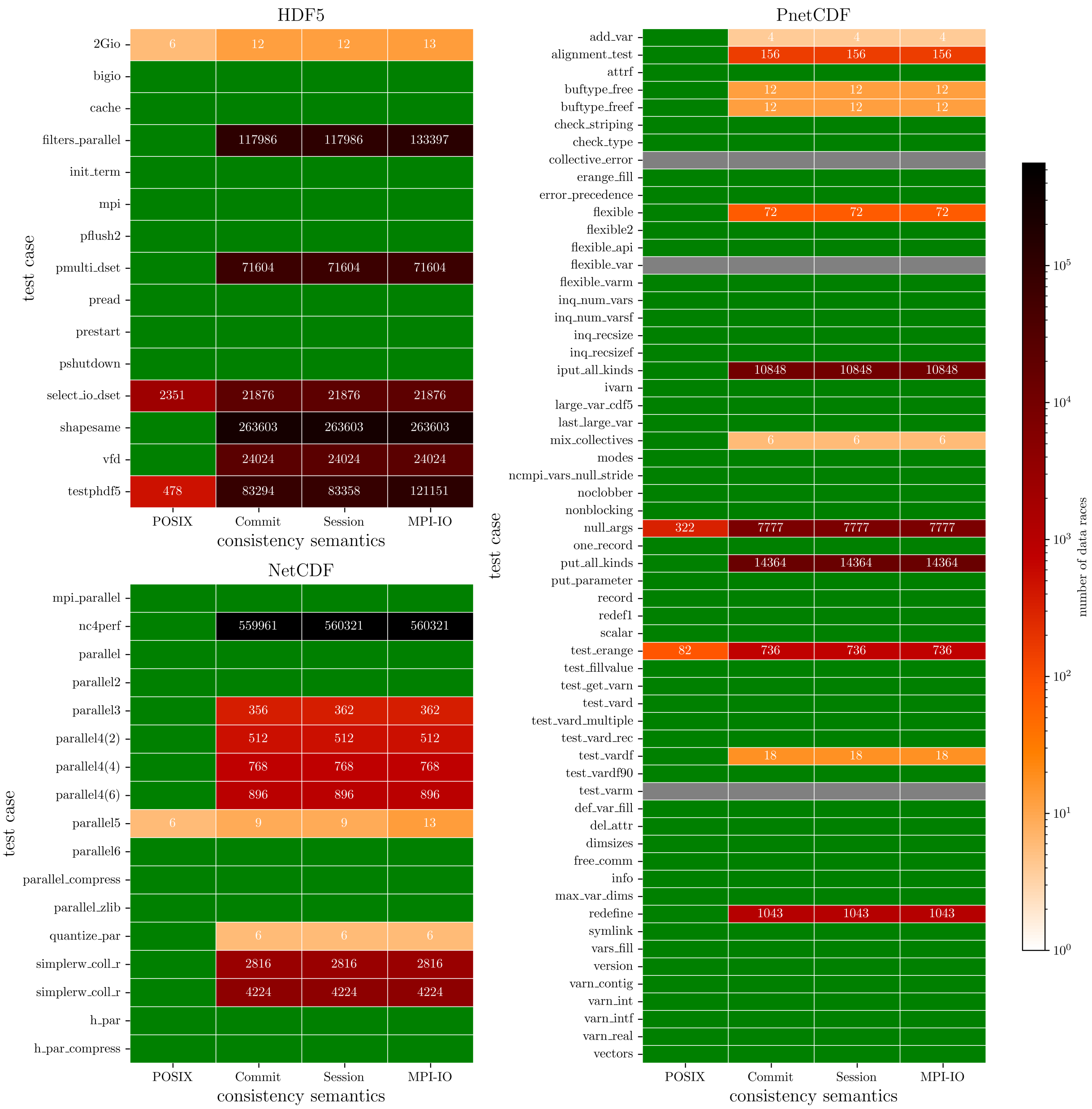


Fig. 4: Number of data races detected. Each row represents a test execution, and each column shows the number of data races under a specific consistency model. Green boxes indicate properly synchronized executions (i.e., zero data races). Gray rows indicate executions where VerifyIO detected unmatched MPI calls, due to potential library implementation issues.

Semantics	HDF5 (15)	NetCDF (17)	PnetCDF (59)	Total (91)
POSIX	3	1	2	6
Commit	7	9	12	28
Session	7	9	12	28
MPI-IO	7	9	12	28

TABLE III: Test executions that are not properly synchronized.

Another unexpected finding relates to the behavior of the libraries under MPI-IO semantics. Since all three libraries use MPI-IO under the hood, we expected no additional data races when moving from POSIX to MPI-IO. However, the results in Fig. 4 and Table III show more data races detected under MPI-IO semantics. This suggests that although these libraries are developed using MPI-IO, they still assume a POSIX file system as the backend, potentially skipping some MPI synchronizations and relying solely on POSIX guarantees. This assumption is risky and can lead to significant correctness issues, which we will explore in Section V-C.

Finally, as we can see from Table III, all three libraries exhibited more synchronization issues under weaker consistency models. Interestingly, VerifyIO reported identical results for the Commit, Session, and MPI-IO models. This suggests that while Session and MPI-IO provide weaker guarantees than Commit, they do not break more applications. This insight is valuable when selecting a weaker consistency model, as it suggests that adopting the weakest model can improve performance without sacrificing application correctness. VerifyIO proves useful in this context, helping to determine the extent to which consistency models can be relaxed while maintaining correctness.

B. POSIX Data Races

POSIX data races are almost always problematic, especially in HPC applications, which are typically deterministic and assume the use of a POSIX-compliant file system. Consequently, such applications should not exhibit data races when running on POSIX systems. Due to space constraints, we focus here on the NetCDF and PnetCDF tests that exhibited POSIX data races. We discuss not only the causes but also whether the responsibility to fix these issues lies with the users or the library developers.

1) NetCDF: The test *parallel5* from NetCDF is the only one that exhibited POSIX data races. These races are largely triggered by the high-level function `nc_put_var_schar`, which writes an entire variable in a single operation. Analyzing the call chain reveals that NetCDF implements this function using a sequence of HDF5 APIs. Specifically, `H5Dwrite` calls `MPI_File_write_at`, which writes the same offset and number of bytes to the same file from different processes, leading to a data race. A review of the test source code shows that it creates a variable with the NC_BYTE type and then writes to it concurrently from multiple processes. This is an incorrect use of `nc_put_var_schar` and should be addressed at the application level, by correcting the test source code itself.

2) PnetCDF: The tests *null_args* and *test_erange* both exhibited POSIX data races due to similar issues. These races occur when multiple processes concurrently invoke `pwrite` to write to the same location in the same file. Tracing the call chain reveals that `pwrite` is called by `MPI_File_write_at_all`, which, in turn, is invoked by `ncmpi_put_var1_text_all` in *null_args.c* and `ncmpi_put_var_uchar_all` in *test_erange.c*. Upon reviewing the source code, we determined that the root cause is that multiple processes attempt to write to the *same variable* without proper synchronization. As with the NetCDF issue, this is not the intended use of these PnetCDF functions. They are designed to be collective calls for writing to *distinct variables*. Therefore, the data races in these cases are also caused by improper usage at the application level and should be corrected by the user.

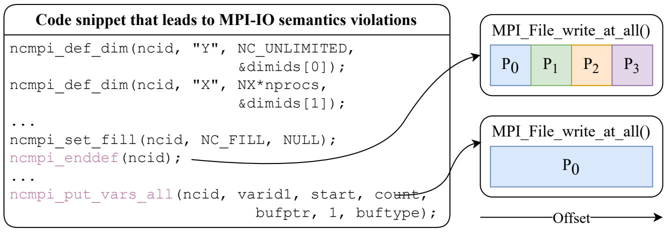
C. MPI-IO Semantics Violations

All three libraries in our study use MPI-IO internally, so they are expected to adhere to MPI-IO semantics. However, as briefly mentioned earlier, this is not always the case. In this subsection, we provide a deeper examination of how PnetCDF and HDF5 violate MPI-IO semantics.

1) PnetCDF: To illustrate the MPI-IO violation in PnetCDF, we analyze the test case *flexible*. Fig. 5 shows a code snippet from *flexible.c* (with some details omitted for readability). The code defines a two-dimensional array variable, initializes it to NULL using `ncmpi_set_fill`, and later populates it with actual values through `ncmpi_put_vara_all`. By examining the call chain, we identified that `MPI_File_write_at_all` is invoked twice internally: first by `ncmpi_enddef` and then by `ncmpi_put_vara_all`. In the first MPI write, each rank writes NULLs to distinct areas of the file. However, before the second write, PnetCDF internally modifies the MPI file view, which triggers MPI-IO's aggregation optimization. As a result, rank 0 performs the entire second write, causing data races between rank 0 and the other ranks.

For typical users, it is difficult to discern which PnetCDF functions are responsible for the actual write or whether they access overlapping file regions. Therefore, expecting users to handle this with appropriate synchronizations is impractical. The issue should instead be resolved at the library level, where there is a complete understanding of when and where actual I/O occurs and how file access patterns change.

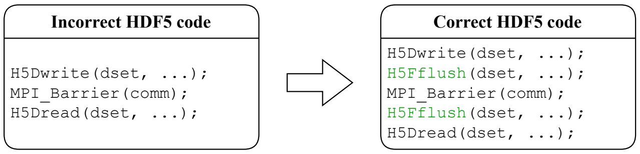
During a review of recent code changes in the PnetCDF tests, we discovered that a workaround had been introduced: `ncmpi_sync/MPI_Barrier/ncmpi_sync` calls were added between potentially conflicting PnetCDF operations. However, these safeguards are only applied on non-POSIX systems and are not enabled by default. This supports our analysis and indicates that the library developers are aware of the issue. Nonetheless, addressing the problem at the application level does not fully resolve the underlying MPI-IO semantics violations within the library's implementation.



```
ncmpi_def_dim(ncid, "Y", NC_UNLIMITED,
                         &dimids[0]);
ncmpi_def_dim(ncid, "X", NX*nprocs,
                         &dimids[1]);
...
ncmpi_set_fill(ncid, NC_FILL, NULL);
ncmpi_enddef(ncid);
...
ncmpi_put_vars_all(ncid, varid1, start, count,
                           bufptr, 1, buftype);
```

Fig. 5: Code snippet from *flexible.c*. In this execution, `MPI_File_write_at_all` is invoked twice, causing a conflict that is not properly synchronized according to MPI-IO semantics.

2) HDF5: We identified a recurring pattern in HDF5 tests (e.g., *shapesame*) that leads to MPI-IO data races. This pattern involves three core functions, shown on the left side of Fig. 6. Specifically, the functions `H5Dwrite` and `H5Dread` access the same dataset, with only an MPI barrier between them. Variations of this pattern exist in other codes, for instance, with `H5Dwrite` and `H5Dread` replaced by other calls, such as `H5Awrite` and `H5Aread`. However, regardless of the specific functions, accessing the same dataset (or attribute) with this pattern consistently leads to a synchronization issue.



```
H5Dwrite(dset, ...);
MPI_Barrier(comm);
H5Dread(dset, ...);
```

```
H5Dwrite(dset, ...);
H5Fflush(dset, ...);
MPI_Barrier(comm);
H5Fflush(dset, ...);
H5Dread(dset, ...);
```

Fig. 6: Improperly vs. properly synchronized HDF5 code

The problem arises because the data returned by `H5Dread` is undefined according to the MPI standard, due to the lack of proper synchronization with the preceding `H5Dwrite`. While the MPI barrier enforces a temporal order, this is only sufficient on systems that provide POSIX consistency. According to MPI-IO semantics, two additional synchronization operations are required, as shown on the right side of Fig. 6. These two `H5Fflush` calls will invoke `MPI_File_sync` to ensure consistency.

However, these flush calls introduce significant overhead, as they force cached data to be written to disk. Since this is unnecessary on POSIX systems, the HDF5 developers (as confirmed in discussions with them) intentionally omit these calls to optimize performance. The assumption is that users will mostly run their code on POSIX systems, where this omission has no impact. Nevertheless, this incorrect synchronization pattern can lead to silent data corruptions when executed on non-POSIX systems, making the issue extremely difficult to detect and resolve.

3) Discussion: At first glance, the MPI-IO violations appear to stem from insufficient MPI synchronizations. However, a deeper examination reveals that this is sometimes an intentional trade-off for performance. In essence, correctness on non-POSIX systems is sacrificed to gain performance on POSIX systems. This reluctance to add MPI synchronizations, in particular `MPI_File_sync`, is primarily due to the high cost of this function. `MPI_File_sync` is designed to ensure both persistency and consistency, though many high-level I/O libraries require only consistency, not persistency. To address this, we have begun an initiative to revise the MPI standard to include a new MPI-IO API that guarantees consistency without forcing a flush to disk. Once this new API is available, high-level libraries will be able to achieve both consistency and improved performance.

D. Potential Implementation Bugs

An additional benefit of VerifyIO is its ability to detect unmatched MPI calls. During its second phase, VerifyIO matches all stored MPI operations, flagging any mismatches or unmatched calls. For example, a collective MPI function must be invoked by all processes within the same communicator; otherwise, VerifyIO will report an error.

In our analysis, we identified three PnetCDF test executions (represented by gray rows in Figure 4) with unmatched MPI calls. The *collective_error* test, as the name suggests, is designed to trigger such errors, and VerifyIO successfully detected this intentional behavior. After manually inspecting the the other two tests, we found the mismatch was caused by the `ncmpi_wait` call. Specifically, during execution, this function splits into two paths: rank 0 calls `MPI_File_write_at_all`, while the other ranks call `MPI_File_write_all`, which appears to be an implementation bug.

E. Overhead and Reproducibility

We conducted an extensive overhead analysis of VerifyIO. Due to space constraints, we present results from the three test cases with the longest verification times. The full set of overhead experiment results is available in VerifyIO's project repository.

Table IV provides a breakdown of execution times for these three cases. Our observations indicate that most of the workflow execution time is spent either on building the happens-before graph or during the verification step. For instance, `pmulti_dset` had the longest runtime, taking approximately 50 minutes to complete. Constructing the happens-before graph, including MPI call matching, took only 69 seconds, resulting in a graph with 119,569 MPI edges and 1,750,025 nodes. The bulk of the time was spent on verification, as it had over 780 million conflict pairs, which were verified sequentially. In contrast, `cache` took around 23 minutes, with nearly all of this time spent on matching MPI calls and building the happens-before graph. This is due to the large number of MPI calls in this test, leading to a final graph with over 16 million nodes and 8 million MPI edges.

It is important to note that some test cases in our evaluation generated significantly larger traces and identified more conflicts than typical real-world applications. In practice, most applications use a limited set of I/O library APIs, make fewer I/O calls, and exhibit fewer conflicts. As a result, we expect VerifyIO to process real-world application traces within a reasonable time frame, if not faster.

Finally, to promote reproducibility and support further research, we have made the source code, collected traces, execution scripts, and all experimental results from this paper publicly available[2].

VI. Related Work

In the context of shared-memory systems, consistency models [20], [21], [22], [23], along with the verification of semantics and program correctness [24], [25], [26], [27], have been extensively studied. However, similar studies in the realm of HPC I/O systems remain sparse. This gap exists despite the increasing complexity and importance of storage consistency in HPC environments.

One reason for this disparity is that HPC I/O stacks lack the compiler layer present in shared-memory systems, which helps programs achieving portability regardless of the underlying consistency models provided by the hardware. As a result, many applications rely on the POSIX [3] interface to ensure

[2] The VerifyIO repository (http://github.com/wangvsa/VerifyIO) includes the source code, execution scripts, and all results presented in this work. The collected trace files can be accessed at https://doi.org/10.5281/zenodo.14553174, and the reproducibility document is available at https://verifyio.readthedocs.io/en/latest/ipdps.html.

	nc4perf (NetCDF)	cache (HDF5)	pmulti_dset (HDF5)
Read Trace	59	20	381
Build the Happens-before Graph	11	1305	69
Generate Vector Clock	3	92	9
Verification	167	0	2608
Total	240	1417	3067

TABLE IV: VerifyIO workflow execution time (seconds) breakdown of the three slowest tests.

portability and compatibility across systems. Given POSIX's dominance in HPC environments, much of the existing work on semantics verification [28], [29], [30] has focused on ensuring program correctness under POSIX semantics.

On the other hand, POSIX enforces strict consistency by ensuring that operations are serialized and immediately visible, but this comes at the cost of performance, especially in distributed environments [11], [31]. To overcome this, modern I/O systems, including parallel file systems [7], [8], [9], [32], [33] and I/O libraries [5], [6], [34], are increasingly adopting weaker consistency models. These models, such as commit consistency, session consistency, and MPI-IO consistency, relax the strict guarantees of POSIX in favor of higher performance and scalability. However, they increase the risk of data races or incorrect behavior in applications that assume POSIX consistency.

Consequently, recent efforts have attempted to address these risks by expanding the scope of consistency checks. AtomFS [35] introduced a formal framework for building verified concurrent file systems, while other work [36] developed a formal model for the Google File System, describing its read/write behaviors and encoding the model in a way that allows automatic verification. Despite these advances, such efforts often focus on a single file system or a specific consistency model, limiting their generality and applicability to other I/O systems and libraries. In parallel, a few projects have explored verification of parallel I/O codes against multiple models. For example, in [37], the authors studied the requirements of consistency semantics of HPC applications and proposed a method for detecting data races for three I/O consistency models. However, the proposed method is based on examining the timestamp of each I/O operation, which is only a proximity of the happens-before order and may result in incorrect results. A follow-up work [38] addresses this shortcoming by examining synchronization events when detecting data races. However, the method remained limited to specific file systems and consistency models, and it only reported the existence of data races without providing detailed information to help developers resolve the issues.

In contrast, VerifyIO addresses these limitations by providing a generic verification framework that supports widely used HPC I/O models. VerifyIO detects data races, identifies mismatched MPI calls, and additionally reports the full call chain associated with detected violations. These features help answer the key questions posed in Section I, offering a more comprehensive solution than existing tools for detecting and diagnosing I/O consistency issues in HPC environments.

VII. Conclusion

In this work, we present a trace-driven verification workflow, implemented in VerifyIO, for determining I/O consistency issues in parallel programs. VerifyIO's tracing tool supports complete API sets of three widely-used I/O libraries, and is easy to extend to support more. The verification tool is capable of handling various consistency models for any given execution trace. Through an extensive study of 91 tests across these I/O libraries, VerifyIO successfully identified consistency issues, providing valuable insights into both application and library-level issues. Finally, all tools in VerifyIO and all collected data and results are made publicly accessible for further study.

To enhance the usefulness of VerifyIO, one area of future work is improving the reported call chain. In many applications, the same function may be called multiple times from different locations, requiring manual inspection during data race analysis. To reduce this manual effort, we plan to explore the integration of a backtrace feature to complement the call chain. Another planned feature is the dynamic selection of the verification algorithm. Since VerifyIO includes four different algorithms, it could dynamically choose the most efficient one based on factors such as the number of conflicts, the size of the happens-before graph, and other relevant metrics.

Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 and was supported by the LLNL-LDRD Program under Project No. 23-ERD-053. LLNL-CONF-870387. This work was supported by NSF SHF Collaborative grant 1763540. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under the DOE Early Career Research Program.

References

[1] SUN, "High-Performance Storage Architecture and Scalable Cluster File System," tech. rep., Sun Microsystems, Inc, 2007.

[2] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST'02, (USA), pp. 231–244, USENIX Association, 2002.

[3] IEEE, "Standard for Information Technology—Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7," *IEEE Std 1003.1, 2013 Edition (incorporates IEEE Std 1003.1-2008, and IEEE Std 1003.1-2008/Cor 1-2013)*, pp. 1–3906, 2013.

[4] J. Li, W.-k. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel NetCDF: A High-Performance Scientific I/O Interface," in *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, (New York, NY, USA), p. 39, Association for Computing Machinery, 2003.

[5] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An Overview of the HDF5 Technology Suite and Its Applications," in *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, AD'11, (New York, NY, USA), p. 36–47, Association for Computing Machinery, 2011.

[6] P. Corbett, D. Feitelson, S. Fineberg, Y. Hsu, B. Nitzberg, J.-P. Prost, M. Snir, B. Traversat, and P. Wong, "Overview Of The MPI-IO Parallel I/O Interface," 1995.

[7] T. Wang, K. Mohror, A. Moody, W. Yu, and K. Sato, "BurstFS: A Distributed Burst Buffer File System for Scientific Applications," in *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[8] M. J. Brim, A. T. Moody, S.-H. Lim, R. Miller, S. Boehm, C. Stanavige, K. M. Mohror, and S. Oral, "UnifyFS: A User-level Shared File System for Unified Access to Distributed Local Storage," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 290–300, IEEE, 2023.

[9] O. Tatebe, S. Moriwake, and Y. Oyama, "Gfarm/BB—Gfarm File System for Node-Local Burst Buffer," *Journal of Computer Science and Technology*, vol. 35, no. 1, pp. 61–71, 2020.

[10] C. Wang, K. Mohror, and M. Snir, "Formal Definitions and Performance Comparison of Consistency Models for Parallel File Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 35, no. 6, pp. 937–951, 2024.

[11] M. Vilayannur, S. Lang, R. Ross, R. Klundt, L. Ward, *et al.*, "Extending the POSIX I/O Interface: A Parallel File System Perspective," tech. rep., Argonne National Lab.(ANL), Argonne, IL (United States), 2008.

[12] D. Kimpe and R. Ross, "Storage Models: Past, Present, and Future," *High Performance Parallel I/O*, pp. 335–345, 2014.

[13] F. Herold and S. Breuner, "An introduction to BeeGFS," tech. rep., ThinkParQ, 2018.

[14] IBM, "Burst Buffer Shared Checkpoint File System," Apr. 2020.

[15] S. Oral, S. S. Vazhkudai, F. Wang, C. Zimmer, C. Brumgard, J. Hanley, G. Markomanolis, R. Miller, D. Leverman, S. Atchley, *et al.*, "End-to-end I/O Portfolio for the Summit Supercomputing Ecosystem," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, 2019.

[16] "MPI: A Message-Passing Interface Standard Version 4.0." https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf, 2021.

[17] R. Rew and G. Davis, "NetCDF: An Interface for Scientific Data Access," *IEEE computer graphics and applications*, vol. 10, no. 4, pp. 76–82, 1990.

[18] C. Wang, J. Sun, M. Snir, K. Mohror, and E. Gonsiorowski, "Recorder 2.0: Efficient Parallel I/O Tracing and Analysis," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, (New Orleans, LA, USA), pp. 1–8, IEEE, 2020.

[19] A. Hagberg, P. J. Swart, and D. A. Schult, "Exploring Network Structure, Dynamics, and Function using NetworkX," tech. rep., Los Alamos National Laboratory (LANL), Los Alamos, NM, USA, 2008.

[20] M. Dubois, C. Scheurich, and F. Briggs, "Memory Access Buffering in Multiprocessors," *ACM SIGARCH computer architecture news*, vol. 14, no. 2, pp. 434–442, 1986.

[21] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," *ACM SIGARCH Computer Architecture News*, vol. 18, no. 2SI, pp. 15–26, 1990.

[22] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors," *Communications of the ACM*, vol. 53, no. 7, pp. 89–97, 2010.

[23] L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, no. 7, p. 558, 1978.

[24] G. Roşu, W. Schulte, and T. F. Şerbănuţă, "Runtime Verification of C Memory Safety," in *International Workshop on Runtime Verification*, pp. 132–151, Springer, 2009.

[25] M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi, "On the Verification Problem for Weak Memory Models," in *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 7–18, 2010.

[26] A. Stefănescu, D. Park, S. Yuwen, Y. Li, and G. Roşu, "Semantics-Based Program Verifiers for All Languages," *ACM SIGPLAN Notices*, vol. 51, no. 10, pp. 74–91, 2016.

[27] A. Bouajjani, E. Derevenetc, and R. Meyer, "Checking and Enforcing Robustness Against TSO," in *Programming Languages and Systems: 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 22*, pp. 533–553, Springer, 2013.

[28] P. Ročkai, Z. Baranová, J. Mrázek, K. Kejstová, and J. Barnat, "Reproducible Execution of POSIX Programs with DiOS," *Software and Systems Modeling*, vol. 20, no. 2, pp. 363–382, 2021.

[29] L. Freitas, J. Woodcock, and A. Butterfield, "POSIX and the Verification Grand Challenge: A Roadmap," in *13th IEEE International Conference on Engineering of Complex Computer Systems (iceccs 2008)*, pp. 153–162, IEEE, 2008.

[30] G. Ntzik, P. da Rocha Pinto, J. Sutherland, and P. Gardner, "A Concurrent Specification of POSIX File Systems," in *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[31] D. Kimpe and R. Ross, "Storage Models: Past, Present, and Future," *High Performance Parallel I/O*, pp. 335–345, 2014.

[32] S. Oral, S. S. Vazhkudai, F. Wang, C. Zimmer, C. Brumgard, J. Hanley, G. Markomanolis, R. Miller, D. Leverman, S. Atchley, and V. V. Larrea, "End-to-end I/O Portfolio for the Summit Supercomputing Ecosystem," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, (New York, NY, USA), Association for Computing Machinery, 2019.

[33] A. Miranda, R. Nou, and T. Cortes, "echofs: A Scheduler-Guided Temporary Filesystem to Leverage Node-local NVMs," in *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 225–228, IEEE, 2018.

[34] H. Tang, S. Byna, F. Tessier, T. Wang, B. Dong, J. Mu, Q. Koziol, J. Soumagne, V. Vishwanath, J. Liu, and R. Warren, "Toward Scalable and Asynchronous Object-Centric Data Management for HPC," in *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '18, p. 113–122, IEEE Press, 2018.

[35] M. Zou, H. Ding, D. Du, M. Fu, R. Gu, and H. Chen, "Using Concurrent Relational Logic with Helpers for Verifying the AtomFS File System," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 259–274, 2019.

[36] B. Li, M. Wang, Y. Zhao, G. Pu, H. Zhu, and F. Song, "Modeling and Verifying Google File System," in *2015 IEEE 16th International Symposium on High Assurance Systems Engineering*, pp. 207–214, IEEE, 2015.

[37] C. Wang, K. Mohror, and M. Snir, "File System Semantics Requirements of HPC Applications," in *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '21, (New York, NY, USA), p. 19–30, Association for Computing Machinery, 2021.

[38] S. Yellapragada, C. Wang, and M. Snir, "Verifying IO Synchronization from MPI Traces," in *2021 IEEE/ACM Sixth International Parallel Data Systems Workshop (PDSW)*, pp. 41–46, IEEE, 2021.