Pinpointing Crash-Consistency Bugs in the HPC I/O Stack: A Cross-Layer Approach

Jinghan Sun
University of Illinois at Urbana-Champaign
Urbana, IL, USA
js39@illinois.edu

Jian Huang
University of Illinois at Urbana-Champaign
Urbana, IL, USA
jianh@illinois.edu

Marc Snir
University of Illinois at Urbana-Champaign
Urbana, IL, USA
snir@illinois.edu

ABSTRACT

We present ParaCrash, a testing framework for studying crash recovery in a typical HPC I/O stack, and demonstrate its use by identifying 15 new crash-consistency bugs in various parallel file systems (PFS) and I/O libraries. ParaCrash uses a "golden version" approach to test the entire HPC I/O stack: storage state after recovery from a crash is correct if it matches the state that can be achieved by a partial execution with no crashes. It supports systematic testing of a multilayered I/O stack while properly identifying the layer responsible for the bugs.

CCS CONCEPTS

• **Information systems** → **Hierarchical storage management**;
• **Software and its engineering** → *File systems management*.

KEYWORDS

Crash consistency, parallel file systems, I/O library

ACM Reference Format:
Jinghan Sun, Jian Huang, and Marc Snir. 2021. Pinpointing Crash-Consistency Bugs in the HPC I/O Stack: A Cross-Layer Approach. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3458817.3476144

1 INTRODUCTION

In order to enable massive I/O parallelism for HPC applications, the HPC community has developed specialized I/O stacks over the past decades [27, 62, 68]. Such stack consists of multiple layers (see Figure 1), including a parallel I/O library such as HDF5 [32] and NetCDF [54] for scientific data management; the MPI I/O library [28] for parallel I/O communication; and a parallel file system (PFS) for distributed data accesses. Much attention was paid to their performance, but few studies investigated their crash consistency – what conditions are satisfied by the storage state of the HPC I/O stack after recovery from a crash [60].

Various crash recovery tools [8–10, 34] are used to restore a system to an internally consistent state after a crash. Fault tolerance techniques such as checkpointing [19, 48] and journaling [18, 47] have also been proposed. Nevertheless, crashes happen frequently, recovery takes a long time, and not all data can be recovered [6]. Recent studies [22, 38] reported that PFSs experienced frequent failures, and their data recovery could take months with manual assistance. Users also repeatedly report losing HDF5 and NetCDF datasets during the past decade [1–5, 7, 11, 12].

Researchers have conducted intensive studies on crash consistency for various file systems [15, 24–26, 43, 50]. They have used two main approaches: formal verification [20, 23, 59, 71] and testing [15, 20, 39, 43, 50, 70–72]. These methods do not apply easily to address the crash-consistency challenges of the HPC I/O stack. It is hard to apply formal methods to production systems, due to their complexity and frequent updates. The developers of an I/O library such as HDF5 cannot test their library without an underlying PFS that is maintained by another vendor. On the other hand, they cannot assume that the underlying PFS is correct. Therefore, they need a testing framework that can test the entire stack as a unit, but still be able to attribute bugs to the proper layer across the I/O stack. Such a framework requires a clear definition of the contract between the I/O layers, i.e., of their *crash consistency model:* Namely, what is a correct state for the I/O layer after recovery from a crash. HDF5 will recover to a correct state, assuming that the PFS recovered to a correct state.

To this end, we present ParaCrash, an effective testing framework for identifying crash-consistency bugs in the HPC I/O stack. We use a "golden master testing" approach, where the tested system is compared to a system assumed to be correct. For crash-consistency bugs, this means comparing the storage state recovered after a crash to a legal reference state occurring without crashes. The legal state is obtained by executing a *legal subset* of I/O operations of the tested system preceding the crash. The legal subsets are defined by a *crash consistency model.* For example, with the strict crash-consistency model, if a crash occurs after a sequence of I/O operations, all the I/O operations preceding the crash need to be recovered. Weaker crash-consistency models allow that some I/O operations to be lost.

ParaCrash runs test programs and traces I/O operations and communications across the stack. It then builds call graph and causality graph to represent the execution order constraints between these operations. After that, it emulates a crash. During the crash emulation, the I/O library invokes MPI and PFS operations that invoke local file system operations up to the crash. The order of the lowest-level storage accesses is only partially constrained by the program logic – file systems may internally reorder I/O operations or lazily flush them to persistent storage. Therefore, the emulation explores different orderings of these accesses, each resulting in a potentially

SC '21, Nov 14–19, 2021, St. Louis, MO

ACM ISBN 978-1-4503-8442-1/21/11…$15.00
https://doi.org/10.1145/3458817.3476144

distinct final crash states. We then run the recovery procedure to obtain a possible recovered state.

On the other hand, a legal state is obtained by running a legal subset of the operations of the tested system before the crash in causality order to completion, including persisting the updates. A legal state for HDF5 is obtained by running a subset of the HDF5 operations; a legal state for the PFS is obtained by running a subset of the PFS operations. The recovered state is compared to the legal states. If there is no match, a crash-consistency bug is found. We then reuse the same procedure for testing the state of the PFS after a crash, comparing it to states obtained from the execution of legal subsets of the PFS operations that preceded the crash. If the PFS state is correct, then the bug is attributed to the I/O libraries atop PFS. Otherwise, it is due to the PFS or the local file system. Since the local file system and MPI-IO are well tested, all the bugs we found were due to either PFS or HDF5.

The procedure of identifying crash-consistency bugs allows using different crash-consistency models at different layers. It can be applied to different multilayered I/O stacks. Its generic approach largely automates the testing procedure. We also provide customized tools for the test programs using specific upper-level I/O library.

We used ParaCrash to test I/O stacks that included HDF5 or NetCDF, MPI-IO, PFS, and local file system ext4. We tested five different PFSs: BeeGFS [37], OrangeFS [61], GlusterFS [21], GPFS [57], and Lustre [58]. ParaCrash identified 15 new crash-consistency bugs, and half of them have been confirmed by developers. For each test, ParaCrash can finish the checking of crash states in half an hour or less. Overall, we make the following contributions:

- We study crash consistency guarantees of a typical HPC I/O stack, and define multiple crash-consistency models for reasoning about its crash-consistency behaviors.
- We develop a new testing approach for the multi-layered, parallel I/O stack. It compares storage states after crash recovery against legal states produced from partial executions allowed by the crash-consistency model.
- We build an automated testing framework named ParaCrash, which can pinpoint the layer responsible for a crash-consistency bug in a typical HPC I/O stack.
- We apply our testing framework to popular PFSs and parallel I/O libraries, and identify 15 new crash-consistency bugs. Some of them are deep consistency bugs that cannot be easily identified without a cross-layer approach.

We release the ParaCrash framework on Github: https://github.com/my-HenryS/ParaCrash.

2 BACKGROUND AND MOTIVATION

In this section, we first present the system architecture of a typical HPC I/O stack, and then discuss the challenges of identifying its crash-consistency bugs.

2.1 System Overview of the HPC I/O Stack

We show the system architecture of a typical HPC I/O stack in Figure 1. PFS often uses local file systems such as ext4 to access disks on each storage server. Other PFSs such as GPFS (a.k.a., Spectrum Scale) directly operate atop the block I/O interface. To aggregate storage capacity and bandwidth, the PFS stripes files across multiple storage servers. The PFS also manages its metadata; metadata is stored either in a database or in a file system and is most often stored on dedicated metadata servers. To facilitate the management of scientific data, the parallel I/O stack usually relies on parallel I/O libraries such as HDF5 [32] and NetCDF [54]. The MPI I/O library is used to optimize the data transfer between application clients and storage servers.

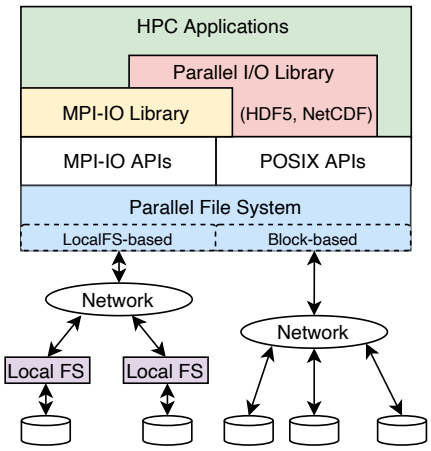

Figure 1: System architecture of HPC I/O stacks.

Compared to generic distributed storage systems, a typical HPC I/O stack has several distinguishing properties. First, it has a deep software stack, with unique components, as shown in Figure 1. Second, the PFS is designed to support efficiently parallel updates to a single file by all processes in a large-scale parallel application. This requires extremely scalable, fine-grained coherence protocols Third, redundancy is usually achieved using RAID storage at the data servers, rather than by replication at the PFS level. Since RAID decreases vulnerabilities to hardware failures, crashes are most often due to the software, rather than the hardware.

2.2 Recovering from a Crash is Challenging

As discussed in Section 1, crashes in the HPC I/O stack are frequent and costly. Therefore, efficient crash recovery is important. The post-crash behaviors of an I/O system depend on what has been persisted before failures. The I/O library stored its state in the PFS and the PFS may have stored its state in local file systems. Therefore we need to recover the local file systems first, followed by the PFS, and then the parallel I/O library. The recovered states of the local file systems may not represent a valid PFS state; and the recovered PFS state may not represent a valid I/O library state.

Crash-consistency models specify what conditions should be satisfied by each of these layers – the contract each layer should obey after recovering from a crash. Each layer will perform its own recovery assuming that the layer below satisfied its contract. Unfortunately, this contract is not clearly defined. POSIX semantics define the behavior of a file system in the absence of crashes, but do not define what can be recovered after a crash [20]. This is also true for HDF5 or other parallel I/O libraries. This ambiguity makes it

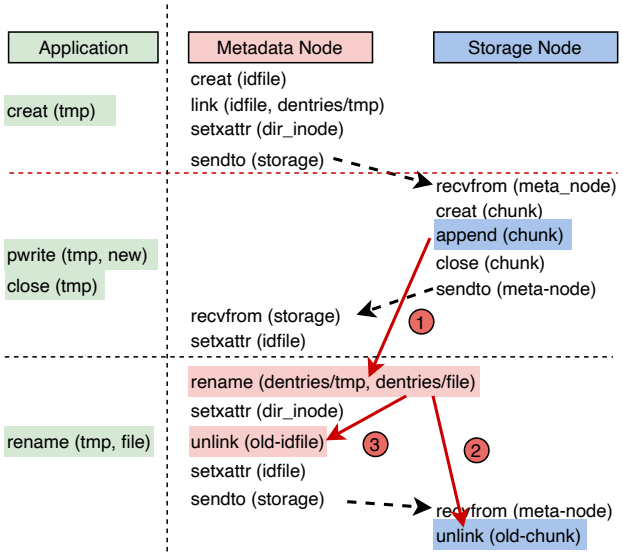

Figure 2: Crash-consistency bugs in BeeGFS. Dashed black arrows indicate server communications. Red arrows indicate possible bugs when the order the updates are persisted is different from the order the updates execute.

harder for I/O library and application developers to conduct crash recovery at their levels.

2.3 Avoiding Consistency Bugs is Challenging

To facilitate our discussion on crash-consistency bugs in the HPC I/O stack, we present an example of BeeGFS in Figure 2. The program attempts to replace the content of a target file by creating, modifying, and renaming a temporary file in BeeGFS.

In this example, we identify three inconsistent crash states: ❶ A crash happens after persisting the `rename` operation for the directory entry on the metadata node, but before persisting the `append` on the storage node. From the application's perspective, both the old and new versions of the `file` will be lost, and `tmp` will be removed. This crash cannot be resolved by the recovery tool `beegfs-fsck` available in BeeGFS. ❷ Similar data inconsistency will happen if a crash happens after persisting the `unlink` operation on the storage node but before persisting the `rename` operation on the metadata node. The `tmp` file contains the updated data, while `file` suffers from data loss. BeeGFS cannot recover `file` data, and this requires application-level recovery mechanism. ❸ The third case is caused by the intra-node persistence reordering between `rename` and `unlink` on the metadata node. For some local file systems such as Btrfs [56], these two directory operations could be reordered. Upon this crash, remounting BeeGFS will cause inconsistent states in its metadata, and thus, the original file cannot be accessed.

These crash-consistency bugs occur for two major reasons: (1) *atomicity violations*, in which updates that are part of an atomic operation are partially persisted; and (2) *ordering violations*, in which updates are not persisted in the order of their execution. In order to avoid these crash-consistency bugs, developers can utilize transactional protocols to enforce atomicity and execute `fsync` calls to enforce ordering. However, this adds significant performance overhead, which limits their use in HPC.

3 RELATED WORK

The HPC I/O stack has been extensively developed over the past decades [21, 37, 57, 58, 61]. Researchers have developed a variety of optimization techniques to improve its performance [53, 67, 74], including the adoption of new storage technologies [64, 65] and high-performance interconnects [66, 69]. Recent studies [22, 29, 36] investigated the fault handling of parallel file systems. However, few studies worked on the crash-consistency issues with a complete HPC I/O stack. To the best of our knowledge, ParaCrash is the first work that conducts a thorough analysis of the crash consistency of a complete HPC I/O stack, and develops an efficient testing framework for pinpointing crash-consistency bugs.

The detection of crash consistency bugs has been studied for local file systems [43] and distributed databases [15]. Several studies applied model checking for identifying crash-consistency bugs [20, 71, 72]. Researchers also utilized fuzzing approaches [39, 70] to detect application-level vulnerabilities [50] and built crash-safe file systems [24, 59]. Specifically, ALICE [50] explored application-level crash consistency based on the analyzed abstract persistence models (APMs) for different local file systems. PACE [15] tested the internal consistency of distributed databases, with a pruning strategy designed for replicated state machines. Both tools require program-specific checker scripts. CrashMonkey [43] could generate test cases and it tested crash consistency for local file systems by comparing crash states against an oracle. It traces block-level I/O and focuses on consistency bugs at persistence points like `fsync`. Unfortunately, none of these tools can be directly applied to the HPC I/O stack. They lack support for parallel programs and do no handle multilayered storage systems. Unlike them, ParaCrash is mainly developed for testing the multilayered HPC I/O stack with a cross-layer approach.

Debugging tools were developed to identify general bugs in Linux file systems. Some of them used fuzzing techniques, such as syzkaller [30] and JANUS [70]. Others utilized static analysis and model checking to identify semantic bugs, such as JUXTA [42] and Recon [33]. However, none of them are specialized for checking crash-consistency bugs, particularly for parallel file systems. ParaCrash aims to detect crash-consistency bugs in the parallel file systems and I/O libraries with enhanced tracing approaches and a new trace analysis framework.

4 PARACRASH DESIGN

In this section, we present ParaCrash, which aims to efficiently test the parallel I/O stack for identifying crash consistency bugs. Given a test program and crash consistency models for each layer, ParaCrash will automatically generate possible crash states, check their correctness, and identify the corrupted I/O layer.

4.1 ParaCrash Overview

We illustrate the system architecture of ParaCrash in Figure 3. ParaCrash takes five steps to check crash-consistency bugs for each test program. First, ParaCrash traces both storage and communication operations on clients and servers, when running a test

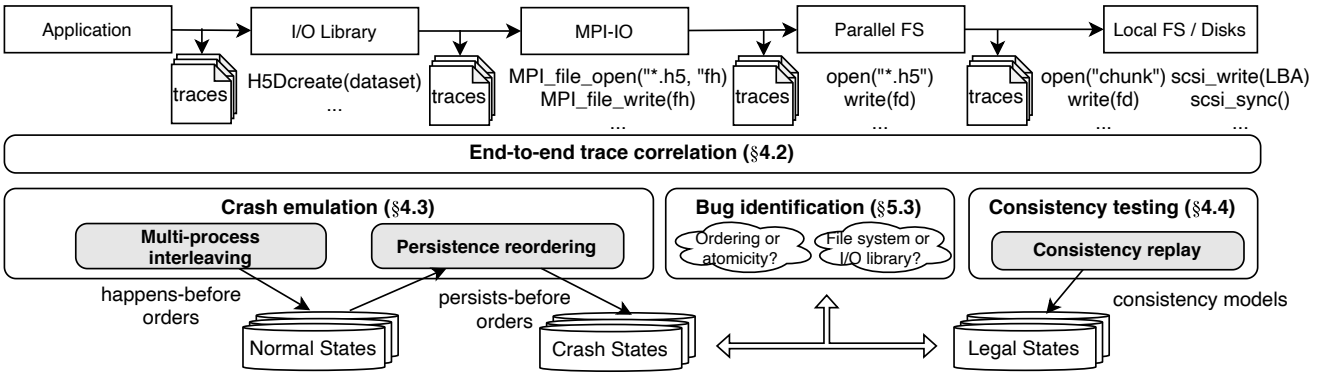

Figure 3: The system architecture of ParaCrash.

program. Second, ParaCrash performs an end-to-end analysis of the traces to associate communication operations to the corresponding storage operations. For each layer, it builds a causality graph that represents the happens-before order of the I/O operations at that layer (§4.2). Third, ParaCrash generates *normal executions* where all the I/O operations up to the crash front were executed, and none after. The operations are executed in an order consistent with the causality ordering. For each of the normal executions, ParaCrash creates *crash states* by dropping non-persisted I/O operations (§4.3). A pruning algorithm is applied to reduce the search space (§5.3). Fourth, ParaCrash executes the system recovery procedure for each crash state, in order to obtain *recovered states*. It also creates *legal storage states* by replaying legal subsets of I/O operations that satisfy the relevant crash consistency model. A recovered crash state is inconsistent if it does not match any of the legal states. The consistency checker will run this test at the highest layer, and if the test fails, it will continue to the lower layers to pinpoint the root cause (§4.4). Finally, ParaCrash removes redundant bugs that have the same root cause, and will classify them based on semantic information (§5.2). We discuss each step in the following sections.

4.2 Trace Recording and Correlated Analysis

ParaCrash builds a multi-layer, multi-process I/O *causality graph* by tracing all necessary storage and communication operations at each I/O layer. This includes the I/O library, MPI and I/O calls at the client and server nodes. For user-level PFSs (e.g., BeeGFS), we trace POSIX I/O operations of the server processes. For kernel-level PFSs (e.g., GPFS) that do not use system calls to interact with the underlying storage layer, we trace block-level I/O commands. The information of each trace entry includes the timestamp and command arguments (e.g., file offset). ParaCrash also traces inter-process communications – both MPI communications at the application level and RPC calls to servers. The MPI calls determine the *happens-before*, or causality order among calls of different clients. The caller-callee relation determines causality across libraries and subsystems; the RPC calls help ParaCrash to order the client events with respect to the server events. The chronological order of client calls is also part of the causality order. This is correct for single-threaded clients, but over-constrained for multi-threaded clients. In order to relax this, we would also need to trace thread synchronizations.

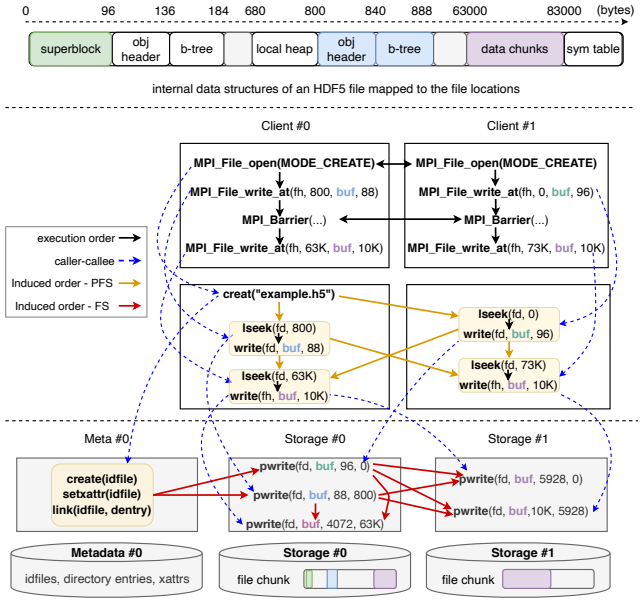

Figure 4: Trace analysis of an HDF5 program.

ParaCrash combines all the separate trace files and creates the causality graph. Its nodes are trace entries and its edges are caller-callee or sender-receiver pairs. In Figure 4, we use an example of MPI-IO file accesses to an HDF5 file stored on PFS to illustrate the described tracing and correlated analysis procedure.

At the bottom I/O layer, we have storage operations that access specific locations in local files or blocks. Our trace analysis can identify which locations are accessed or modified in persistent storage for each I/O library function call, or for each PFS call. When the layout of the internal data structures of the I/O library or the PFS is known, we can additionally associate the file locations and file offsets with the data structures of the I/O library. Figure 4 shows the mapping from PFS files to local files. Using this mapping and the recorded arguments of the write calls (e.g., `MPI_File_write`, `pwrite`), each call will be tagged with the data structures it modifies. While not necessary for bug detection, this information can be used to prune the state search space and benefit our root cause analysis.

Algorithm 1: Crash State Generation

```
Input: The causality graph of lowermost-level I/O operations
Output: crash_states
journaling ← data, ordered or writeback
normal_states ← all consistent cuts of the causality graph
foreach normal_state ∈ normal_states do
    lfs_ops ← lowermost-level I/O operations before the cut
    // pick 0-k victims that will not persist, along with
       their dependent operations
    foreach n ∈ [0, k] do
        choices_of_victims ← combinations(lfs_ops, n)
        unpersisted ← {}
        foreach victims ∈ choices_of_victims do
            foreach victim ∈ victims do
                unpersisted += depends_on(victim, lfs_ops)
            crash_state ← lfs_ops − unpersisted
            crash_states.append(crash_state)
/* Find operations that depend on the victim                */
Function depends_on(victim, lfs_ops):
    dependencies ← {victim};
    foreach lfs_op ∈ lfs_ops − victim do
        foreach d ∈ dependencies do
            if persists_before(d, lfs_op) then
                dependencies.append(lfs_op)
    return dependencies
```

Algorithm 2: Local FS Persistence dependency

```
/* Check if op1 should persist before op2                   */
@lru_cache;
Function persists_before(op1, op2):
    if same_localfs(op1, op2) then
        if journaling = data then
            return happens_before(op1, op2)
        else if journaling = writeback then
            return is_meta(op1)     is_meta(op2)
              happens_before(op1, op2)
        ...
    else
        return ∃ sync : sync.fd = op1.fd and
          happens_before(op1, sync) and
          happens_before(sync, op2);
```

4.3 Crash Emulation

ParaCrash emulates possible crash states by replaying the I/O operations traced at the lowermost level, i.e., POSIX I/O calls to the local file systems or the block-level I/O operations. Replaying I/O operations at the lowermost level will expose errors induced by any of the upper-level I/O layers. We do not need to execute the operations in the order of their timestamps: any total order that is consistent with the causality order will be explored, as it corresponds to a possible execution.

We present the pseudocode for our crash state exploration in Algorithm 1. At the beginning, the crash emulator creates normal execution states by exploring all possible *consistent cuts*. A *cut* splits the I/O operations into BEFORE (the crash) and AFTER (the crash) sets. The cut is *consistent* if no I/O operations in AFTER happens before an I/O operation in BEFORE. The storage state obtained by executing all the BEFORE set and none of the AFTER set of a consistent cut is defined as a *normal state*. A *crash state* is obtained by picking up to k victim I/O operations from the BEFORE set and not persisting them, where k is a configurable parameter. The happens-before order of the execution imposes some constraints on the *persist-before* order of persist operations, as explained below. When a victim I/O operation is not persisted, some other dependent I/O operations will not be persisted either. Finally, the algorithm creates a list of possible crash states at the local storage level. Each I/O layer above will have its view of the crash states by reading from the lower I/O layers.

To determine which operations are not persisted, the algorithm computes the persist-before order for I/O operations. For operations on the same local file system, the persistence order depends on the enabled journaling mode in the local file system. With data journaling, operations are persisted in the order they execute; hence the persistence order among operations on the same local file system is same as the causality order. The operations that were persisted at a local file system up to a crash could be any prefix of the sequence of I/O operations executed by the local server. Any combination of such prefixes determines a possible global crash scenario. With a relaxed journaling mode like writeback mode [52], only metadata operations are ordered.

For I/O commands to the block device, their ordering are enforced by barriers. Once we determine the persistence ordering, crash states are explored in a similar way to local file systems.

For local file system operations or block I/O commands to different storage servers, their relative order is jointly constrained by commit operations (e.g., `fsync` and `scsi_synchronize_cache`) and the causality order. If the normal execution contains a commit operation, all updates to the same file that preceded the commit call are persisted.

For each selection of crash states, ParaCrash emulates their updates on the snapshot of the initial local file system or the image of the block device, and then restarts and remounts the PFS. The number of possible crash states can be large, and many distinct crash states could expose the same bug. Therefore, we develop various empirical pruning techniques to reduce the search space (see the detailed discussion in Section 5.3).

4.4 Consistency Checking

ParaCrash tests crash consistency with respect to consistency models. A *crash consistency model* specifies what conditions should be satisfied by a storage system after recovery from a crash. It reflects the persistence expectations that an upper-level storage system or an application holds. Unfortunately, these are not defined by existing standards, nor are they documented for products. Yet, without such definitions, it is impossible to decide which layer is responsible for a failure to recover to an acceptable state after a crash. This is why precise definitions of crash consistency models are essential in multilayered storage systems. They are more important in the context of HPC, as I/O libraries must work with many different underlying PFSs, and a focus on performance leads to weaker guarantees for crash recovery. In this section, we propose a general

approach to defining crash consistency models, propose a few specific models that we believe are adequate for HPC I/O, and describe how ParaCrash tests an HPC I/O stack with these models.

4.4.1 Storage System Semantics. We derive crash consistency models from the behavior of the storage system in the absence of crashes. Therefore, the first step is to define what these behaviors should be. The POSIX standard [35] provides an informal specification for POSIX compliant file systems. Formal specifications have been provided by various studies [17, 31, 44, 45, 55]. For our purposes, suffice to specify that some operations are required to be atomic, in particular, metadata updates. And the operations should appear to execute in the order they were issued: a read returns the value of the last preceding write to the same location. When I/O operations are issued by multiple processes, precedence should be understood to be the causality or "happens-before" partial order determined by the execution order of each process, and the order of synchronizations between processes [40]. The same definitions can be used for an I/O library, such as HDF5: metadata operations are expected to be atomic, and causality order is preserved.

4.4.2 Correct Crash Recovery. Ideally, a storage system should support "precise exceptions": if the user code is single-threaded, after recovery, we expect that all the I/O operations before the crash point appear to have completed, and no I/O operations after the crash point appear to have started. For multi-threaded or multi-process user codes, the failure "point" is a consistent cut through the causality graph: after recovery, all I/O operations preceding the consistent cut appear to have completed, and none after the consistent cut appear to have started. Non-atomic storage operations concurrent with the crash could be only partially executed.

Unfortunately, it is expensive to support a precise exception model. Therefore, weaker models are proposed. We postulate that recovery semantics for a storage system should be defined in terms of the API of that system, irrespective of how the system is implemented. This leads to the following approach: A correct recovery protocol should bring the storage to a state that could be obtained by executing a subset of the I/O operations preceding the crash in their original causality order. We call this partially ordered subset a *preserved set*. Different consistency models will correspond to different definitions of the legal preserved sets. For the HPC I/O stack, we consider the following crash-consistency models:

Strict Crash Consistency. As discussed, it means that all operations preceding the crash, and only those, appear to have executed. If the program is deterministic, then the storage is restored to the state at the crash point (i.e., one preserved set per crash); if it is non-deterministic, it is restored to the state that could have been achieved with a different multiprocessing schedule.

Commit Crash Consistency. Storage systems often provide commit operations that ensure that preceding I/O operations are persisted to the storage. A typical example is POSIX `fsync`: `fsync` flushes dirty data associated with a file to persistent storage. Commit crash consistency requires that (1) if a commit operation (e.g., `fsync(fd3)` in Figure 5) happened before the crash, then all I/O operations persisted by this commit (e.g., `write(fd3, "C")` in Figure 5) are in the the preserved set; (2) operations that were not persisted by the commit operations but happened before the crash

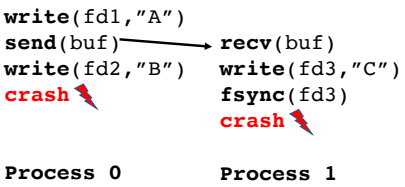

Figure 5: A parallel execution at the point of a system crash.

could be either persisted or not persisted, leading to different preserved sets; (3) operations that happen after the crash are not in the preserved set.

Causal Crash Consistency. This model extends the commit crash consistency with an additional requirement: if an I/O operation *A* is in the preserved set, then any I/O operation that happened before *A* is also in the preserved set. The definition of causal crash consistency seems to match the intuition of programmers, and is a good match for the dominant I/O pattern of scientific applications, where the computation goes through successive phases, each depending on the previous one. A local file system such as ext4, mounted in data journaling mode, is causally consistent. Parallel file systems usually recover to a causally consistent state even though they do not explicitly guarantee it.

Baseline Crash Consistency: We believe that the weakest reasonable crash consistency model is to require that if a file or a dataset was closed before the crash occurred, then all updates to that file before it was closed were preserved. Formally, the preserved set includes all updates performed on a file or dataset that was not opened in write mode when the crash happened.

We illustrate these definitions in Figure 5. With strict crash consistency, all three writes are preserved. With commit crash consistency, the write of C is preserved, but the writes of A or B may be lost. With causal crash consistency, the writes of A and C are preserved, but it is legal for the write of B to be lost. With baseline consistency, all three writes may be lost.

4.4.3 Crash-Consistency Checking. Once a crash state is created for the checked I/O layer, ParaCrash will check whether it satisfies the required crash consistency model, i.e., whether it matches a legal "golden state" generated by replaying a legal preserved set of I/O operations. To reduce the checking overhead, the checker first checks the storage consistency using the checker available for the corresponding I/O layer. For example, we have `beegfs-fsck` for BeeGFS, `mmfsck` for GPFS, and `h5check` for HDF5. These tools check that the internal storage system structure is consistent, but they do not check which I/O operations preceding the crash were preserved. If the first consistency test passes, we then test the crash state against the specified consistency model. To do so, ParaCrash compares each recovered crash state to the legal states obtained by the replay of corresponding preserved sets of I/O operations.

If a crash state fails to match any of the legal states, we run recovery tools (e.g., `h5clear` for HDF5) to resolve the inconsistency. We only regard this state as inconsistent if the recovery tools cannot fix the inconsistency issue. The choice of a crash consistency model leads to different preserved sets and hence different legal storage states (see Section 4.4.2). Since weaker models allow for a larger number of legal states, fewer behaviors are considered to be wrong.

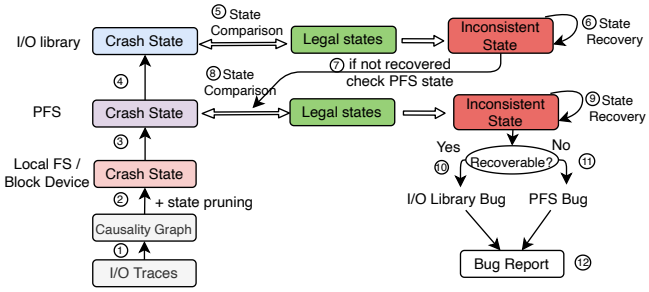

Figure 6: The workflow of ParaCrash. We use circled numbers to indicate the order of the workflow.

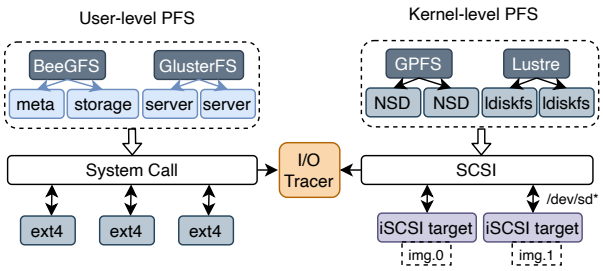

Figure 7: I/O tracing approach for PFS in ParaCrash.

ParaCrash checks crash states using a top-down approach: it checks whether the crash state is a correct I/O library state, based on the legal preserved sets of I/O library calls. If not, it checks whether the crash state is a correct PFS state, based on the legal preserved sets of PFS operations. If the state is an invalid I/O library state, but a valid PFS state, then the bug is attributed to the I/O library; otherwise it is attributed to the PFS. We illustrate the entire workflow of ParaCrash in Figure 6.

5 IMPLEMENTATION AND OPTIMIZATIONS

ParaCrash takes a configuration file and two programs as input, and automatically generates crash-consistency reports for the tested I/O stack. The configuration file is used to specify (1) system configurations like the PFS mount point, storage directories, file stripe size, number of servers and client; (2) the crash consistency model for each I/O layer, and (3) the mode used for crash state exploration. ParaCrash supports three crash-state exploration modes: brute-force, pruning, and optimized exploration. The brute-force mode implements Algorithm 1 and 2 to generate crash states. The pruning mode leverages an efficient crash state pruning mechanism to cut down the exploration space. And the optimized exploration mode enables incremental crash state exploration in addition to pruning. The two programs are a preamble program that initializes the storage system and a test program that runs next. ParaCrash traces and emulates crashes for the test program. In this section, we will discuss implementation details and ParaCrash optimizations.

5.1 Tracing and Replaying

The I/O tracing component of ParaCrash is developed based on the `Recorder` tracing tool [63], `strace` [14], and Open-iSCSI [41, 73]. We use `Recorder` and `strace` to trace the HDF5, MPI-IO and I/O calls of all the test programs. For user-level PFS (e.g., BeeGFS), we use `strace` to trace the local I/O operations. For kernel-level PFS (e.g., GPFS), we mount them on iSCSI disks and trace block-level I/O commands. As shown in Figure 7, the PFS I/O tracing component is generic. The syscall-based tracing applies to the user-level PFS and SCSI-based tracing applies to the kernel-level PFS. ParaCrash also traces communication calls at each layer. Similar to I/O tracing, we use `Recorder` for tracing MPI communications and `strace` for communications between user-level PFS servers. For kernel-level like GPFS, we trace their tcp communications between servers.

We enhanced `Recorder` in order to trace all relevant I/O operations (including HDF5, MPI-IO, and POSIX) of the test program. We implement these new features using 253 additional LoC to trace additional HDF5 and MPI functions, and record more detailed function call parameters. For example, ParaCrash records both the address and content of the `MPI_File_write` call. It also records the exact `memspace` and `filespace` dimension of `H5Dwrite`. The tracing component of ParaCrash is lightweight, and it does not require program modification. After tracing, a separate file will be created for each process with traces at each I/O layer.

To generate crash states and legal storage states, ParaCrash replays I/O traces at different I/O layers. To replay operations of the parallel or local file systems, ParaCrash utilizes the Python's `OS` module to invoke system calls. Replaying I/O calls at the HDF5 level requires a different approach. We build an `h5replay` tool to replay HDF5 I/O operations. Given a sequence of HDF5 operations to replay, it creates a C program containing the HDF5 function calls and their dependent statements, and executes the generated program.

5.2 Bug Classification

ParaCrash uses the rules described in Table 1 to classify bugs. We classify a bug as a reordering issue between two operations if the first pattern is observed (Table 1a), and as an atomicity issue if the second pattern (Table 1b) is observed. ParaCrash also checks atomicity issues for more than two operations.

ParaCrash generates a crash consistency report for each test program based on the I/O layer the crash is attributed to and its bug classification. As multiple test programs may expose similar bugs, we use the following rules to aggregate inconsistent crash states that have the same cause. For test programs using POSIX APIs, two inconsistent crash states are considered to have the same cause, if they are caused by the broken atomicity or reordering of the same pair of operations. For test programs using an I/O library like HDF5, we also take the I/O library objects (e.g., B-tree nodes and symbol table nodes) into count. In this case, two inconsistent crash states are considered to have the same cause, if they involve the same set of operations on the same data or metadata structures.

We implemented a tool named `h5inspect` to map HDF5 objects to their location in the file, which will assist with the bug classification and multi-level trace analysis in §4.2. Given an HDF5 file, `h5inspect` generates a JSON file to record its object mapping information. We implement `h5inspect` based on `h5check` [34], and it supports HDF5 1.8, HDF5 1.10, and NetCDF 4.7.

Table 1: Pairwise failure patterns. "✗" mark denotes that the crash state fails the test, while "✓" mark indicates that it passes the test.

Inconsistency I_1		O_B	
		Persisted	Not persisted
O_A	Persisted	✓	✓
	Non-persisted	✗	✓

(a) Reordering issue: O_A should be persisted before O_B.

Inconsistency I_2		O_B	
		Persisted	Not persisted
O_A	Persisted	✓	✗
	Non-persisted	✗	✓

(b) Atomicity issue: O_A should be persisted with O_B.

5.3 Optimizations for State Exploration

The crash state exploration procedure may generate hundreds or even thousands of possible crash states for a simple test program. The reconstruction of a crash state can take several seconds, even for a small-scale PFS configuration of four servers. Therefore, this can lead to a long exploration time for each single test program. To address these two issues, we propose two optimization techniques:

To address the state explosion challenge, we develop an efficient *crash state pruning* mechanism to reduce the number of crash state reconstructions. Specifically, ParaCrash maintains a list of inconsistent crash scenarios that have been explored, and skips scenario that has been already explored.

Consider the configurations described in Table 1a, where O_A and O_B are the lowermost I/O operations traced at different servers (local filesystem I/O operations for user-level PFS; block I/O operations for kernel-level PFS). If a recovered state fails the test when O_A was not persisted but O_B was, but any of the other three combinations pass the test, then we identify that the problem is due to a reordering of these two operations. Henceforth, we do not test scenarios where O_A is not persisted while O_B is persisted. The pruning rule for avoiding duplicate atomicity issue is specified in a similar manner: Once we find that several operations have to be executed atomically, we do not further test scenarios where these operations are partially persisted.

In addition, we leverage the semantic information obtained from the mapping of high-level data structures to low-level local file offsets to further prune the exploration space. For example, storage operations that update data chunks in the I/O library dataset will not be reordered, as they are not likely to incur metadata consistency issues. We show in Section 6.4 that our proposed pruning mechanisms can greatly reduce the number of crash states, while not decreasing test coverage.

To reduce the state reconstruction time, ParaCrash implements an incremental crash state reconstruction mechanism. In the brute-force mode, we create each crash state by replaying all the local file system I/O operations. PFS has to synchronize with its local file system, which is usually accomplished by restarting the PFS. However, the differences between crash states can be merely one or two I/O operations, and we do not need to restart from scratch when moving from one state to another. An optimized order in which crash states are visited is chosen to reduce the number of PFS rebooting operations. We model this procedure as solving a traveling salesman problem (TSP). The collection of all crash states are the nodes of an undirected graph, and the distance between two crash states is the number of PFS servers in different states. In ParaCrash, we use a greedy TSP solver [13] to find an optimized visiting path.

Table 2: System Configuration.

Category	Software Version	Configuration
Operating System	Ubuntu 16.04	default
	CentOS 7.9	lustre-patched
Parallel File System	BeeGFS 7.1.2	tuneRemoteFSync
	OrangeFS 2.9.7	default
	GlusterFS 5.13	striped volume
	GPFS 5.0.4	default
	Lustre 2.12.6	default
Parallel I/O Library	MPICH 3.0.4	shared library, pvfs enabled
	HDF5 1.8.12	shared library, mpi enabled
	NetCDF 4.7.5	shared library, mpi enabled

6 PARACRASH EVALUATION

To evaluate the efficiency of ParaCrash, we run test programs on various parallel file systems and investigate the causes and the consequences of the identified crash consistency bugs. ParaCrash found 15 new crash consistency bugs and attributed them to the PFS or the parallel I/O library. We also show that our optimized exploration mechanisms are effective, as they achieved up to 12.6× performance improvement while identifying the same bugs.

6.1 System Configuration

We evaluate ParaCrash with five parallel file systems: GPFS (a.k.a., Spectrum Scale) [57], Lustre [58], BeeGFS [37], OrangeFS [61], formerly known as PVFS2, and GlusterFS [21]. Their configurations are shown in Table 2. BeeGFS, OrangeFS, and Lustre are configured with two metadata servers and two storage servers. For GlusterFS and GPFS, we run two servers in total, as each server can manage both data and metadata. The stripe size at each storage server is set to 128KB. When a file grows larger than the stripe size, its file chunks are stored across data servers in a round-robin manner. As for the metadata services, BeeGFS and GlusterFS store their metadata with extended attributes, while OrangeFS uses the default Berkeley DB [46] to store the metadata information.

We use HDF5 (h5py as the Python wrapper) and NetCDF (pynetcdf4 as the Python wrapper, HDF5 format) as the parallel I/O libraries in our evaluation, as HDF5 and NetCDF are deployed in a majority of HPC systems as their core I/O libraries. The caching of parallel I/O libraries is enabled by default. The PFS servers for BeeGFS, OrangeFS, and GlusterFS are assumed to run their own local file system ext4. The local file system uses its safest mode in our evaluation – data journaling. Following the testing approach used in previous research [15], ParaCrash enables PFS testing with multiple servers on a single machine, with each server being a separate process or runs in individual virtual machine listening on a distinct network port.

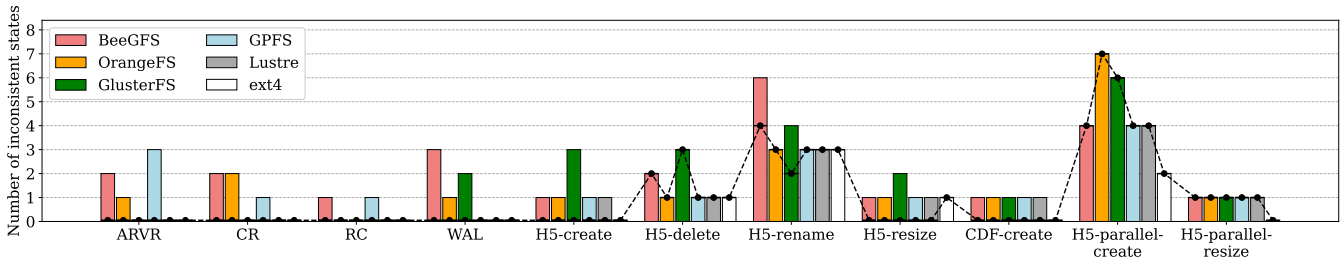

Figure 8: Number of inconsistent crash states on different file systems. The line plots represent the number of HDF5 inconsistencies where PFS is in the correct states.

6.2 Test Cases and Test Models

We use 11 representative test programs to evaluate ParaCrash, including POSIX-IO programs, HDF5 and NetCDF programs, and parallel HDF5 programs. For all test programs, we test PFS with the causal crash consistency model, since none of the parallel file systems satisfies strong consistency, but all of them satisfy causal consistency. For HDF5 and NetCDF programs, the I/O libraries are tested with both baseline consistency and causal crash consistency.

We follow the common practice of using the simplest test for identifying a bug. The test programs use code fragments found in real HPC programs. If we identify a bug using these test programs, the bug could exist in real HPC programs as well. ParaCrash allows users to generate their own test programs, although the generation of test programs is not the focus of this work. As we will discuss in Section 6.3, simple test programs can reveal many crash-consistency bugs across the HPC I/O stack.

POSIX Programs. Each POSIX program invokes a sequence of I/O system calls to the parallel file system. The test program typically consists of two or more key operations. We list these test programs and their initial states as follows. Some of them were used in previous file system studies [20, 51].

- Atomic-Replace-via-Rename (ARVR). This program atomically updates the file content of a preexisting file `foo`. ARVR creates a temporary file `tmp`, writes to it with the updated content, and then replaces the original file with the `tmp` via a rename operation. This pattern is used by checkpointing libraries [16, 49] to ensure that the latest checkpoint file always has the same name.
- Create-and-Rename (CR). The PFS is initialized with two directories `A` and `B`. The CR program creates a file `foo` under `A` and moves it to the other directory.
- Rename-and-Create (RC). The PFS is initialized with one directory named `A`. The RC program renames it to `B` and next creates a file `B/foo`.
- Write-Ahead-Logging (WAL). The PFS is initialized with a file `foo`. The WAL program first writes `logs` to record file modifications, and then it overwrites the file content with multiple pages.

HDF5 and NetCDF Programs. Each program contains one or two common I/O library function calls, such as dataset creation or deletion. These programs start with a common initial state in which a file stores two groups and two datasets. The fundamental HDF5 operations include dataset creation, deletion, rename, and resize. Correspondingly, our HDF5 test suite tests each of these operations with test programs `H5-create`, `H5-delete`, `H5-rename`, and `H5-resize`. As NetCDF supports fewer operations on variables and groups, we only test its variable create and rename operation with `CDF-create` and `CDF-rename`. We do not find any bugs with `CDF-rename`, so we do not report it in this paper. We also have two parallel programs in the evaluation, each of them runs collective HDF5 calls (i.e., dataset creation and resize). We believe these programs represent common use cases, as they cover the essential metadata operations of these I/O libraries.

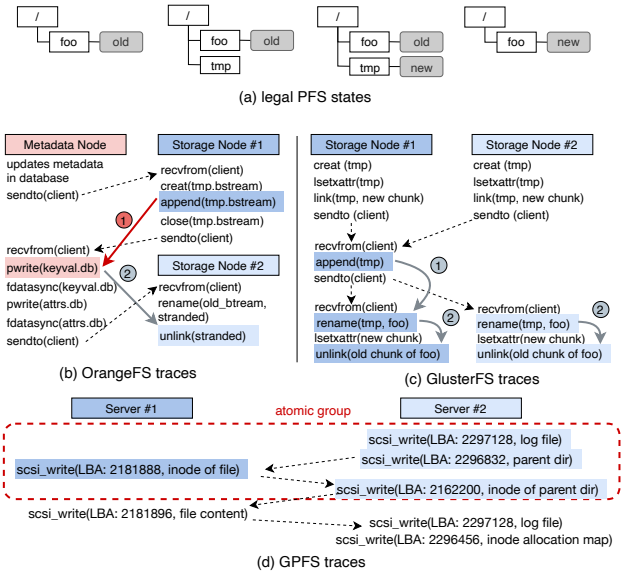

Figure 9: ARVR program crashes differently on BeeGFS (see Figure 2), OrangeFS, GlusterFS, and GPFS. The legal storage states under causal consistency are shown in figure (a). Figure (b) and (c) show that some BeegFS bugs do not occur in OrangeFS and GlusterFS. Figure (d) shows GPFS traces. GPFS will be inconsistent if the atomic group of writes is partially persisted.

Sensitivity. We observe that whether consistency bugs are triggered may depend on the system configuration. With POSIX programs, PFS accesses multiple files and directories across the servers. Their file distribution pattern may affect the persistence ordering between I/O operations. Therefore, we test POSIX programs with different distribution patterns and check if they expose different bugs. HDF5 and NetCDF programs use tree structures to

Table 3: List of crash consistency bugs discovered by ParaCrash. We show in the 4th column the details of each bug. We use *op@server* to represent an operation executed on the server and # to distinguish between different servers of the same type. We use $A \rightarrow B$ to show that A should be persisted before B, and $[A, B]$ to indicate that A should be persisted with B. In the 3rd column, the root cause layer is the same as the inconsistent layer if we do not specify it.

No.	Program	Inconsistent layer (Root Cause Layer)	Details	Consequence	Sensitivity
1.	ARVR	BeeGFS, OrangeFS	append(file chunk of `tmp`)@storage → rename(d_entry of `tmp`, d_entry of `foo`)@metadata	Data loss	N/A
2.	ARVR	BeeGFS	rename(d_entry of `tmp`, d_entry of `foo`)@metadata → unlink(old file chunk of `tmp`)@storage	Data loss	N/A
3.	ARVR	GPFS	**[**write(log file)@server#2, write(parent_dir)@server#2, write(file inode)@server#1, write(parent_dir inode)@server#2**]**	Data loss (accept all mmfsck fixes) Metadata loss (if inode entry not deleted)	N/A
4.	CR	BeeGFS, OrangeFS	link(idfile, d_entry of `A/foo`)@metadata → unlink(d_entry of `B/foo`)@metadata	File created in both directories	N/A
		GPFS	write(inode of directory `A/`)@server → write(inode of directory `B/`)@server		
5.	RC	BeeGFS	rename(d_entry of `A`, d_entry of `B`)@metadata#1 → link(idfile, d_entry of `B/foo`)@metadata#2	File created in a wrong directory	file distrib.
		GPFS	write(directory entry of the parent directory)@server → write(log file)@server		N/A
6.	WAL	BeeGFS, GlusterFS, OrangeFS	append(file chunk of `log`)@storage#1 → overwrite(file chunk of `foo`)@storage#2	No logs written after file modification	file distrib.
7.	WAL	BeeGFS	link(idfile, d_entry of `log`)@metadata → overwrite(file chunk of `foo`)@storage	No logs created after file modification	N/A
8.	WAL	BeeGFS, GlusterFS	overwrite(file chunk of `foo`)@storage → unlink(d_entry of `log`)@metadata	No logs created after file modification	N/A
9.	H5-parallel-create	HDF5	Local heap → B-tree nodes of the same group	Cannot open an unmodified dataset	# of clients
10.	H5-create	HDF5 (BeeGFS, OrangeFS, GlusterFS, GPFS, Lustre)	B-tree nodes & local name heap → Symbol table node of the same group	Cannot open an unmodified dataset	N/A
11.	H5-delete	HDF5	Symbol table node → B-tree nodes & local heap of the same group	Cannot open an unmodified dataset	N/A
12.	H5-rename	HDF5	**[**B-tree nodes, symbol table & local heap from both source and destination group**]**	The renamed dataset is lost	N/A
13.	H5-parallel-resize, H5-resize	HDF5 (BeeGFS, OrangeFS, GlusterFS, GPFS, Lustre)	Superblock → B-tree node of the resized dataset	Cannot read data from the resized dataset (addr overflow)	h5clear options
14.	H5-resize	HDF5	Child B-tree node → Parent B-tree node	Cannot read data from the resized dataset (wrong B-tree signature)	dim. of dataset
15.	CDF-create	NetCDF (BeeGFS, OrangeFS, GlusterFS, GPFS, Lustre)	Superblock → Object header	Cannot open the file (NetCDF: HDF5 error [Errno -101])	N/A

organize metadata (B-tree nodes and symbol table nodes). Inconsistencies could be triggered when the nodes in the tree are split across servers. Thus, we test them with a variety of dataset dimensions (from 200×200 to 1000×1000, 200×200 by default in our setting), and different numbers of datasets per group (from 1 to 8, 2 by default). For parallel programs, we also choose a different number of clients (from 1 to 10, 2 by default). We report our sensitivity study results in Table 3. Note that increasing the number of servers, or the number of victims in Algorithm 1, did not expose new bugs.

6.3 Identified Crash-Consistency Bugs

For each test program, ParaCrash reports the inconsistent crash states and attributes each bug to either HDF5 or PFS. We remove the redundant inconsistent states and identify their unique root causes (see Section 5.2).

We report the test results of ParaCrash in Figure 8. ParaCrash identified 15 unique crash-consistency bugs. We list them in Table 3.

6.3.1 PFS Crash Consistency Bugs. As expected, we found that PFSs suffer from more crash-consistency bugs than local file systems. As shown in Figure 8, as we replace the PFS with ext4 that uses data journaling mode, no crash leaves the system in an inconsistent state for any of the POSIX test programs.

For BeeGFS, we find that each of the POSIX I/O test programs may lead to inconsistent states after crashes. The bugs are results of the improper synchronization of persistent I/O operations across different servers. For example, the two crash-consistency bugs illustrated in Figure 2 are exposed by the ARVR program. Every POSIX test program identified crash-consistency bugs in BeeGFS. Their consequences include data loss, files created in wrong location, and the loss of log files.

OrangeFS provides stronger persistence guarantees for its metadata services, since each 4KB modification to the database will be followed by a `fdatasync()` operation (see Figure 9). This prevents the reordering of I/O operations on the metadata servers, and thus helps OrangeFS prevents one data inconsistency scenario for test program ARVR. However, ParaCrash still identified crash consistency bugs in OrangeFS. They occurred for two reasons: (1) multiple updates on the metadata server are not issued in the correct order, and the `fsck` of OrangeFS fails to fix the inconsistency (e.g., CR program); (2) I/O operations on distinct storage servers could be persisted in different order than how they are issued. Therefore, the `append` operation in the ARVR program could still be reordered after the directory entry rename (implemented by database updates) and this results in data loss if the system crashes between their persistence.

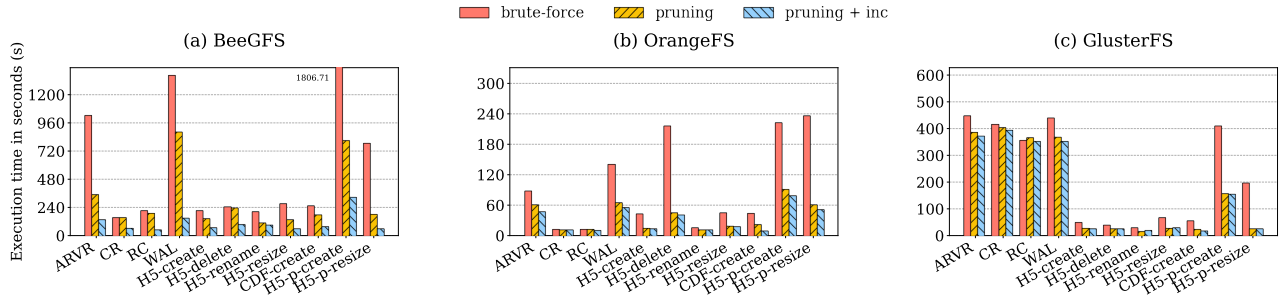

Figure 10: Performance of ParaCrash using different crash state exploration strategies.

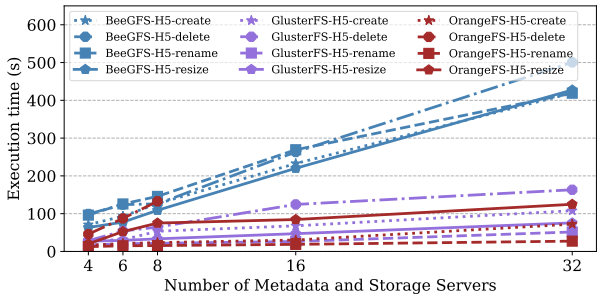

Figure 11: Scalability of ParaCrash as we increase the number of storage servers. [1]

GlusterFS does not have dedicated metadata servers. The metadata and data chunks of a single file or directory are stored on the same servers. The metadata and data operations to a single file are ordered when the file is smaller than the stripe size, as these operations are executed by the same local file system. This prevents all vulnerable reorderings of the ARVR program. However, crash-consistency bugs can happen, when data updates are issued to different files or different stripes of a single file, across multiple storage servers. ParaCrash identified such bugs (see Figure 8). For example, the WAL test program that modifies two files, and HDF5 programs that operates against large files.

ParaCrash also reports bugs in kernel-level PFS – GPFS and Lustre. GPFS suffers from data inconsistency after running three out of the four POSIX programs. For CR and RC test program, GPFS bugs have similar causes and consequences as for BeeGFS. As shown in Figure 9(d) and Table 3, the root cause and consequence of running ARVR program against GPFS is different, and we treat it as a new bug. The recovery mechanism of file systems affects the consequence severity. After we run the GPFS checker mmfsck, GPFS still suffers from data or metadata loss. We did not find crash-consistency bugs in Lustre when we run the POSIX test programs. This is because Lustre properly aggregates intermediate changes to the files and invokes accurate disk barriers to flush data to the disk. However, we identified crash-consistency bugs in both Lustre and GPFS, when we run HDF5 test programs. We will describe them in the following section.

6.3.2 HDF5 and NetCDF Crash Consistency Bugs. After testing the I/O stack with ParaCrash, we find that HDF5 violates baseline crash consistency (`H5-create`, `H5-delete`, `CDF-create`, and `H5-parallel-create`) as well as causal consistency (`H5-resize`, `H5-rename`). All these identified bugs can result in application data loss. The violations of baseline crash consistency make unmodified datasets inaccessible. The bugs exposed by `H5-resize` and `H5-rename` corrupt the resized or renamed dataset. We confirmed these bugs with HDF5 developers. We also confirmed with h5py developers that these bugs do not come from the python interface.

We list all the bugs (whether attributed to the PFS or HDF5 library) in Table 3. As shown, a majority of crash-consistency bugs at the HDF5 level are caused by the improper persistence ordering of I/O operations against the library's metadata structures. For example, the parent B-tree node and its child node in the `H5-resize` program should be persisted in the correct order. This crash-consistency bug happens as we increase the dataset dimension from 800×800 to 1000×1000.

6.3.3 Cross-layer Bug Attribution. We attribute bugs to the PFS if it violates the causal crash consistency. This is the case for the bugs exposed by the tests of `H5-create`, `H5-resize`, `H5-parallel-resize`, and `CDF-create`. All other bugs are attributed to the HDF5 library. Proper bug attribution depends on the assumed contract between HDF5 and PFS: The bugs attributed to HDF5 will affect HDF5 running atop other PFSs that are not strongly crash consistent – essentially all of them. On the other hand, if the PFS only commits to satisfy a weaker consistency model, then some of its crash states will become legal, and bugs attributed to the PFS could be attributed to HDF5. This is the virtue of integrated testing: it forces an agreement on the respective crash handling responsibilities of the two layers.

.

6.4 ParaCrash Performance

ParaCrash can efficiently explore crash-consistency bugs with test programs. We compare our proposed optimization strategies with our brute-force exploration as the baseline. Since increasing k (the number of victims in Algorithm 1) did not expose new bugs, we use $k = 1$ to report the performance of our crash state exploration. We evaluate the performance of ParaCrash using BeeGFS, OrangeFS, and GlusterFS. Generating all possible crash states in a brute-force manner may take up to 1806 seconds. We show in Figure 10 that our optimized exploration strategies are effective. In this experiment, we run all test programs with their default settings.

[1] H5-delete program fails as we run it on OrangeFS with more than eight storage servers. This is due to a bug in OrangeFS that crashes the file system when `ftruncate()` is invoked. We have reported this bug to the OrangeFS developers.

With our proposed pruning mechanism, the crash state exploration time for each test program running on different file systems is significantly reduced (up to 2.9× for POSIX programs and 7.3× for HDF5 programs). For example, the exploration time of ARVR program on BeeGFS is significantly reduced from 1021.5 seconds to 352.4 seconds, as the pruning decrease the number of crash states from 280 to 99. The pruning strategy reduces the total number of crash states by 2.2× on average. As discussed in Section 5.3, the pruning mechanism is especially effective for HDF5 programs, in which we leverage the semantic information from the object mappings to skip many crash states.

With incremental crash-state exploration, ParaCrash reduces the time of generating each crash state by 4.2× on average. This optimization is especially effective for testing BeeGFS, as it requires the longest time (up to 7.8 seconds) to restart the PFS. Together with the pruning, our incremental exploration mechanism reduces the time of testing BeeGFS by 5.0× on average. Note that these optimization strategies did not reduce the number of bugs discovered. We obtain similar performance trends as we run ParaCrash with GPFS and Lustre.

To evaluate the scalability of ParaCrash, we increase the number of storage servers. In this experiment, we decrease the PFS stripe size as the number of servers increases. For example, we choose the default 128KB stripe size for all 4-server settings and 16KB for 32-server settings (BeeGFS requires stripe size > 64KB, so we do not further decrease it). We show the scalability result in Figure 11. Since the file is striped into more chunks, there are more combinations of I/O operations persisted on different servers. Therefore, without pruning, the exploration time will increase exponentially as we increase the number of servers. ParaCrash offers better scalability: Its execution time increases linearly as we increase the number of servers. Note that we do not identify new crash-consistency bugs as we increase the number of servers or clients. This is expected, since our tests, especially those related to the metadata of parallel file systems, involve dependencies between a moderate number of storage operations. Therefore, running ParaCrash at a larger scale may not be required.

7 CONCLUSION

We take an initial effort in studying the crash-consistency bugs in the HPC I/O stack. In order to properly attribute bugs to the responsible I/O layer, we provide a clear definition of crash-consistency models for the entire I/O stack, which could guide the future development of parallel file systems and I/O libraries. To pinpoint crash-consistency bugs in the HPC I/O stack, we develop a testing framework named ParaCrash, which employs a golden master testing methodology and various optimization techniques. Our evaluation shows that ParaCrash efficiently identifies 15 new bugs in the popular parallel file systems and I/O libraries.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their helpful comments and feedback. We also thank Weiwei Jia for his help with the setup of the Lustre file system. This work was partially supported by NSF grant CCF-1763540, CNS-1850317, and CCF-1919044.

REFERENCES

[1] Recover a corrupt HDF5 file. https://forum.hdfgroup.org/t/recover-a-corrupt-hdf5-file/1146, 2008.
[2] HDF5 file state in case of crash. https://forum.hdfgroup.org/t/hdf5-file-state-in-case-of-crash/1598, 2010.
[3] Preventing file corruption on crash. https://forum.hdfgroup.org/t/preventing-file-corruption-on-crash/2462, 2012.
[4] Recovering from power loss and timeline for journaling. https://forum.hdfgroup.org/t/recovering-from-power-loss-and-timeline-for-journaling/3293, 2014.
[5] Corrupt files when creating HDF5 files without closing them (h5py). https://stackoverflow.com/questions/31287744/corrupt-files-when-creating-hdf5-files-without-closing-them-h5py, 2015.
[6] HPCC power outage event at Texas Tech. http://www.ece.iastate.edu/~mai/docs/failures/2016-hpcc-lustre.pdf, 2016.
[7] Avoiding corruption of the HDF5 file. https://forum.hdfgroup.org/t/avoiding-corruption-of-the-hdf5-file/4087, 2017.
[8] FSCK: an online file system checker for Lustre. https://github.com/Xyratex/lustre-stable/blob/master/Documentation/lfsck.txt, 2017.
[9] pvfs2-fsck – check and correct file system errors. https://www.mankier.com/1/pvfs2-fsck, 2017.
[10] BeeGFS file system check (beegfs-fsck). https://www.beegfs.io/wiki/FSCheck, 2018.
[11] Avoiding a corrupted HDF5-file or be able to recover it. https://forum.hdfgroup.org/t/avoiding-a-corrupted-hdf5-file-or-be-able-to-recover-it/5441, 2019.
[12] Recovering NetCDF files after data loss. https://www.unidata.ucar.edu/support/help/MailArchives/netcdf/msg14595.html, 2019.
[13] Greedy, suboptimal solver for the travelling salesman problem. https://pypi.org/project/tsp-solver2/, 2020.
[14] strace - trace system calls and signals. https://man7.org/linux/man-pages/man1/strace.1.html, 2020.
[15] Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Correlated crash vulnerabilities. In *Proccedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, Savannah, GA, 2016.
[16] Jason Ansel, Kapil Arya, and Gene Cooperman. Dmtcp: Transparent checkpointing for cluster computations and the desktop. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12. IEEE, 2009.
[17] Konstantine Arkoudas, Karen Zee, Viktor Kuncak, and Martin Rinard. Verifying a file system implementation. In *International Conference on Formal Engineering Methods*, pages 373–390. Springer, 2004.
[18] Remzi H. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*, volume 42. Arpaci-Dusseau Books LLC, 2017.
[19] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. PLFS: a checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12. IEEE, 2009.
[20] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Specifying and checking file system crash-consistency models. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*, Atlanta, GA, April 2016.
[21] Eric B Boyer, Matthew C Broomfield, and Terrell A Perrotti. Glusterfs one storage server to rule them all. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2012.
[22] Jinrui Cao, Om Rameshwar Gatla, Mai Zheng, Dong Dai, Vidya Eswarappa, Yan Mu, and Yong Chen. PFault: A general framework for analyzing the reliability of high-performance parallel file systems. In *Proceedings of the 2018 International Conference on Supercomputing*, pages 1–11. ACM, 2018.
[23] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*, 2017.
[24] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. Using crash Hoare logic for certifying the FSCQ file system. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*, Monterey, CA, 2015.
[25] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP'13)*, 2013.
[26] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency without ordering. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*, 2012.

[27] Avery Ching, Kenin Coloma, Jianwei Li, Alok Choudhary, and Wei keng Liao. High-performance techniques for parallel I/O. http://users.eecs.northwestern.edu/~choudhar/Publications/ChiCol07.pdf, 2001.

[28] Peter Corbett, Dror Feitelson, Sam Fineberg, Yarsun Hsu, Bill Nitzberg, Jean-Pierre Prost, Marc Snir, Bernard Traversat, and Parkson Wong. Overview of the MPI-IO parallel I/O interface. In *Proceedings of the Workshop on Input/Output in Parallel and Distributed Systems (IPPS'95)*, 1995.

[29] Dong Dai, Om Rameshwar Gatla, and Mai Zheng. A performance study of lustre file system checker: Bottlenecks and potentials. In *Proceedings of 35th International Conference on Massive Storage Systems and Technology (MSST'19)*, Santa Clara, CA, 2019.

[30] David Drysdale. Coverage-guided kernel fuzzing with Syzkaller. *Linux Weekly News*, 2:33, 2016.

[31] Gidon Ernst, Gerhard Schellhorn, Dominik Haneberg, Jörg Pfähler, and Wolfgang Reif. Verification of a virtual filesystem switch. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 242–261. Springer, 2013.

[32] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. An overview of the hdf5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, pages 36–47, 2011.

[33] Daniel Fryer, Kuei Sun, Rahat Mahmood, TingHao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. Recon: Verifying file system consistency at runtime. *ACM Transactions on Storage (TOS)*, 8(4):1–29, 2012.

[34] The HDF Group. h5check: the hdf5 format checker. https://support.hdfgroup.org/products/hdf5_tools/h5check.html, 2014.

[35] The Open Group. POSIX.1-2008, IEEE Std 1003.1™-2017 (Revision of IEEE Std 1003.1-2008), Base Specifications Issue 7. http://pubs.opengroup.org/onlinepubs/9699919799/.

[36] Runzhou Han, Duo Zhang, and Mai Zheng. Fingerprinting the checker policies of parallel file systems. In *2020 IEEE/ACM Fifth International Parallel Data Systems Workshop (PDSW)*, pages 46–51. IEEE, 2020.

[37] Jan Heichler. An introduction to BeeGFS. http://www.beegfs.de/docs/whitepapers/Introduction_to_BeeGFS_by_ThinkParQ.pdf, 2014.

[38] Michael P. Kasick, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. Black-box problem diagnosis in parallel file systems. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10)*, San Jose, CA, February 2010.

[39] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*, Ontario, Canada, October 2019.

[40] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558, 1978.

[41] Kalman Z Meth and Julian Satran. Design of the iscsi protocol. In *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, 2003.(MSST 2003). Proceedings.*, pages 116–122. IEEE, 2003.

[42] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 361–377, 2015.

[43] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, pages 33–50, 2018.

[44] Carroll Morgan and Bernard Sufrin. Specification of the UNIX filing system. *IEEE Transactions on Software Engineering*, (2):128–142, 1984.

[45] Gian Ntzik, Pedro da Rocha Pinto, Julian Sutherland, and Philippa 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.

[46] Michael A Olson, Keith Bostic, and Margo I Seltzer. Berkeley DB. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191, 1999.

[47] Sarp Oral, Feiyi Wang, David Dillow, Galen Shipman, Ross Miller, and Oleg Drokin. Efficient object storage journaling in a distributed parallel file system. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10)*, 2010.

[48] Xiangyong Ouyang, Raghunath Rajachandrasekar, Xavier Besseron, Hao Wang, Jian Huang, and Dhabaleswar K Panda. Crfs: A lightweight user-level filesystem for generic checkpoint/restart. In *2011 International Conference on Parallel Processing*, pages 375–384. IEEE, 2011.

[49] Simon Pickartz, Niklas Eiling, Stefan Lankes, Lukas Razik, and Antonello Monti. Migrating linux containers using criu. In *International Conference on High Performance Computing*, pages 674–684. Springer, 2016.

[50] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, Broomfield, CO, 2014.

[51] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Crash consistency. *Communications of the ACM*, 58(10):46–51, 2015.

[52] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *Proceedings of 2005 USENIX Annual Technical Conference (USENIX'05)*, 2005.

[53] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. Indexfs: Scaling file system metadata performance with stateless caching and bulk insertion. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 237–248. IEEE, 2014.

[54] Russ Rew and Glenn Davis. NetCDF: an interface for scientific data access. *IEEE computer graphics and applications*, 10(4):76–82, 1990.

[55] Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. SibylFS: formal specification and oracle-based testing for POSIX and real-world file systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 38–53, 2015.

[56] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: the linux b-tree filesystem. *TOS*, 9(3):9:1–9:32, 2013.

[57] Frank B Schmuck and Roger L Haskin. GPFS: A shared-disk file system for large computing clusters. In *FAST*, volume 2, 2002.

[58] Philip Schwan et al. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux symposium*, volume 2003, pages 380–386, 2003.

[59] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*, 2016.

[60] Jinghan Sun, Chen Wang, Jian Huang, and Marc Snir. Understanding and finding crash-consistency bugs in parallel file systems. In *Proceedings of the 12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'20)*, 2020.

[61] OrangeFS Team. The OrangeFS project. http://www.orangefs.org/, 2015.

[62] Tiankai Tu, Charles A. Rendleman, Patrick J. Miller, Federico Sacerdoti, Ron O. Dror, and David E. Shaw. Accelerating parallel analysis of scientific simulation data via Zazen. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10)*, 2010.

[63] Chen Wang, Jinghan Sun, Marc Snir, Kathryn Mohror, and Elsa Gonsiorowski. Recorder 2.0: Efficient parallel I/O tracing and analysis. In *The IEEE International Workshop on High-Performance Storage*, 2020.

[64] Teng Wang, Sarp Oral, Yandong Wang, Brad Settlemyer, Scott Atchley, and Weikuan Yu. Burstmem: A high-performance burst buffer system for scientific applications. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 71–79. IEEE, 2014.

[65] Teng Wang, W Yu, K Sato, A Moody, and K Mohror. Burstfs: A distributed burst buffer file system for scientific applications. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2016.

[66] Md Wasi-ur Rahman, Xiaoyi Lu, Nusrat Sharmin Islam, Raghunath Rajachandrasekar, and Dhabaleswar K Panda. High-performance design of YARN mapreduce on modern HPC clusters with Lustre and RDMA. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 291–300. IEEE, 2015.

[67] Sage A Weil, Kristal T Pollack, Scott A Brandt, and Ethan L Miller. Dynamic metadata management for petabyte-scale file systems. In *SC'04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, pages 4–4. IEEE, 2004.

[68] Wikipedia. Parallel I/O. https://en.wikipedia.org/wiki/Parallel_I/O, 2020.

[69] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar Panda. PVFS over InfiniBand: Design and performance evaluation. In *2003 International Conference on Parallel Processing, 2003. Proceedings.*, pages 125–132. IEEE, 2003.

[70] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland'19)*, San Francisco, CA, May 2019.

[71] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: A lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*, 2006.

[72] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems (TOCS)*, 24(4):393–423, 2006.

[73] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S Yang, Bill W Zhao, and Shashank Singh. Torturing databases for fun and profit. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, pages 449–464, 2014.

[74] Qing Zheng, Kai Ren, Garth Gibson, Bradley W Settlemyer, and Gary Grider. Deltafs: Exascale file systems scale better without dedicated servers. In *Proceedings of the 10th Parallel Data Storage Workshop*, pages 1–6, 2015.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We ran experiments on local machines with Intel Core i7-7700K CPU @ 4.20GHz, 16G DRAM, and 256GB SSDs. The operation system is Ubuntu 18.04 and we use ext4 as the local filesystem. We set up BeeGFS (beegfs-7.1.2) with multi-mode, OrangeFS (orangefs-2.9.7) with MPICH support, and GlusterFS (gluster-5.13) with striped volume. We build HDF5 library with shared and enable-parallel options. We detect consistency bugs for HDF5 (h5py 2.10.0, HDF5 1.8.12) and netCDF (py-netcdf4 1.5.3, netcdf 4.7.4). We use netCDF4 format (based on HDF5) for netCDF library. ParaCrash is implemented with Python 3.7.5 and GCC 7.5.0.

Author-Created or Modified Artifacts:

```
Persistent ID: https://github.com/my-HenryS/ParaCrash
Artifact name: ParaCrash
Citation of artifact: 10.5281/zenodo.5168471
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: Intel Core i7-7700K CPU @ 4.20GHz

Operating systems and versions: Ubuntu 18.04

Compilers and versions: GCC 7.5.0; MPICH 3.0.4; Python 3.7.5

Applications and versions: HDF5 (h5py 2.10.0, HDF5 1.8.12) and netCDF (py-netcdf4 1.5.3, netcdf 4.7.4)

Libraries and versions: MPICH 3.0.4;