Techniques for Modeling Large-Scale HPC I/O Workloads

Shane Snyder, Philip Carns, Robert Latham, Misbah Mubarak, Robert Ross
Argonne National Laboratory
Argonne, IL, USA
ssnyder@mcs.anl.gov

Christopher Carothers
Rensselaer Polytechnic Institute
Troy, NY, USA

Babak Behzad, Huong Vu Thanh Luu
University of Illinois at Urbana-Champaign
Urbana, IL, USA

Surendra Byna, Prabhat
Lawrence Berkeley National Laboratory
Berkeley, CA, USA

ABSTRACT

Accurate analysis of HPC storage system designs is contingent on the use of I/O workloads that are truly representative of expected use. However, I/O analyses are generally bound to specific workload modeling techniques such as synthetic benchmarks or trace replay mechanisms, despite the fact that no single workload modeling technique is appropriate for all use cases. In this work, we present the design of *IOWA*, a novel I/O workload abstraction that allows arbitrary workload consumer components to obtain I/O workloads from a range of diverse input sources. Thus, researchers can choose specific I/O workload generators based on the resources they have available and the type of evaluation they wish to perform. As part of this research, we also outline the design of three distinct workload generation methods, based on I/O traces, synthetic I/O kernels, and I/O characterizations. We analyze and contrast each of these workload generation techniques in the context of storage system simulation models as well as production storage system measurements. We found that each generator mechanism offers varying levels of accuracy, flexibility, and breadth of use that should be considered before performing I/O analyses. We also recommend a set of best practices for HPC I/O workload modeling based on challenges that we encountered while performing our evaluation.

1. INTRODUCTION

I/O workload modeling is a critical but frequently overlooked aspect of storage system simulation and evaluation. Storage systems must be studied in the context of appropriate workloads to ensure that the evaluation is both relevant and accurate, particularly HPC storage systems that are often subjected to large-scale, coordinated I/O workloads. Various methods are available for capturing I/O workloads from real-world systems, including tracing tools [29, 31, 34, 43], server and storage device instrumentation [17,21,25] and application characterization [9,36,42]. Various methods are also available for reproducing I/O workloads, including trace replay tools [29, 31, 43, 44], synthetic benchmarks [1, 6, 24], synthetic workload generators [20, 23], and application I/O kernels [15,19,40]. Each method offers distinct tradeoffs; no one technique works best in all scenarios.

PMBS2015, November 15-20 2015, Austin, TX, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4009-0/15/11 ...$15.00
DOI: http://dx.doi.org/10.1145/2832087.2832091

In this work we present a generic I/O workload abstraction layer that interchangeably supports diverse sources of large-scale HPC I/O workload data for use in storage system simulations, I/O replay engines, and other I/O evaluation and analysis tools. We leverage this workload abstraction layer in the context of multiple I/O workload consumers because we observe that each lends itself to specific classes of I/O evaluations—simulations allowing researchers to evaluate new storage designs using relevant workloads and I/O replays allowing researchers to directly examine the impact of some workload on a specific storage system implementation, for instance. This abstraction layer also enables researchers to select the appropriate source of workload information depending both on the type of evaluation that they wish to perform and on the data that is available. We evaluate and contrast three distinct methods for generating I/O workloads in this framework using the following sources of workload information.

- **I/O trace workloads**: I/O traces provide highly detailed information regarding each I/O operation issued by a traced application, including timing information and I/O parameters. These traces can be used to reproduce the exact I/O pattern exhibited by the original application. Recorder [29] is an example tracing tool that traces HPC applications at multiple layers of the I/O stack.

- **Synthetic I/O workloads**: Synthetic I/O workloads are manually developed I/O workload descriptions used to impose some desired I/O pattern on a storage system. As an example, the CODES I/O language [27] allows researchers to model real or hypothetical I/O workloads using domain-specific language constructs.

- **I/O characterization workloads**: I/O characterizations provide high-level statistics such as access sizes and interarrival times for application I/O operations, rather than complete traces. Darshan [9] is one such tool that provides I/O characterizations that may be used to derive representative I/O workloads for a given application.

The primary contributions of this paper are the design and demonstration of a modular I/O workload abstraction layer that supports numerous I/O representations, an evaluation of the strengths and weaknesses of three distinct modeling methodologies, and a recommended set of best practices for I/O researchers to use when modeling HPC I/O workloads. We also provide preliminary design details and performance results for an innovative technique for synthesizing representative I/O workloads from Darshan I/O characterizations.

This paper is organized as follows. Section 2 provides background information on each of our three target sources for representative I/O workload information, as well as details regarding the CODES storage simulation framework. In Section 3, we explain the design of our workload abstraction API and describe the implementation of the workload generators we analyze in this study. Section 4 demonstrates two potential use cases for the workload abstraction. In Section 5, we provide some best practices for modeling HPC I/O workloads based on challenges we encountered in the process of analyzing our workload generation techniques. In Section 6, we discuss prior research in the area of I/O workload modeling and storage system simulation. In Section 7, we summarize our findings and provide some potential avenues for future research in this area.

2. BACKGROUND

2.1 I/O event tracing with Recorder

One way to capture I/O workloads from real-world systems is *event tracing*. Event tracing tools trace program functions of interest and capture detailed information about them such as their parameters, timing information, and return value. Recorder [29], a multilevel tracing framework, is an I/O tracing tool that works at multiple layers of the parallel I/O stack, namely, the HDF5, MPI-IO, and POSIX layers.

Recorder has both a static and a dynamic library that may be linked to a given application (preloaded at runtime in the case of the dynamic library). Whenever an MPI process calls an I/O function that is instrumented at a specific layer of the I/O stack by Recorder, the timestamp, function name, arguments, return value, and the duration of the function are stored into a per-process trace file. Analysis tools can then inspect these trace files directly in order to extract high-resolution details of the traced application's I/O workload.

2.2 I/O workload description with the CODES I/O language

The CODES storage simulation framework (Section 2.4) includes a domain-specific language that decouples HPC I/O workload models from the simulations that execute them [27]. The CODES I/O language can express aggregate I/O patterns as well as independent I/O operations for each process. It also provides synchronization (barrier), conditional, and loop constructs to aid in the composition of complex sequences of I/O operations. A parser library converts CODES I/O language files into streams of events for consumption by arbitrary I/O analysis tools (which have traditionally been limited to CODES storage system models). The primary advantage of the CODES I/O language is that it allows users to describe arbitrary synthetic I/O patterns at any scale.

2.3 I/O characterization with Darshan

Darshan is a lightweight, application-level I/O characterization tool designed to capture I/O access pattern information from HPC applications with minimal overhead [9]. Darshan does not log a complete record of each I/O function call and its parameters. Instead, it gathers compact histograms, cumulative timers, and statistics that represent salient properties of I/O behavior. This information is recorded independently at each process on a per-file basis and stored in a bounded amount of memory. Darshan defers all communication and storage activity until the application is shutting down, at which time it aggregates shared file records, compresses remaining file records in parallel, and writes the results to a single, compact log file using collective I/O.

Although Darshan does not provide the fidelity of a traditional I/O tracing tool, its lightweight design makes it suitable for full-time deployment on production systems. Data collected with Darshan has been used to perform broad studies of systemwide I/O trends [8] and to classify production applications according to I/O behavior [10]. The ALCF I/O Data Repository [7] provides public access to Darshan characterizations of hundreds of thousands of HPC jobs spanning a variety of scientific domains. An ability to generate I/O models based on Darshan logs therefore has the potential to provide access to a broad sampling of HPC applications.

2.4 CODES simulation framework

CODES [13] is a storage simulation framework for exploring the design space of exascale storage systems. It can be used to evaluate storage algorithms and storage architectures as well as application I/O performance. CODES is built on top of ROSS [11], a high-performance parallel discrete event simulation system that has been shown to process billions of events per second [3] at scale. ROSS achieves this massive event rate primarily through its optimistic simulation strategy [12], where events are speculatively executed and rolled back in the case of causality violations. CODES is then able to leverage this high-performance ROSS framework to simulate exascale storage architectures and compute resources with high fidelity while still producing timely results.

CODES is organized as a collection of component models that have been modularized to simplify validation and facilitate reuse across a variety of storage system models. For example, CODES includes four different network interconnect models that can be enabled at runtime and accessed through a consistent API. A model configuration language is used to describe how model components are connected, mapped to simulation processes, and configured. Previous simulation studies have successfully utilized a synthetic language to describe I/O workloads generated by compute nodes [27], but a generalized I/O workload component and API are the next logical steps toward enabling a broader variety of simulation scenarios.

3. WORKLOAD GENERATION TECHNIQUES

3.1 I/O workload abstraction: IOWA

In this section we propose the design of IOWA, a novel I/O workload abstraction for generating workloads from a range of diverse input sources, allowing researchers more flexibility in the types of evaluations they may perform. We outline the following design criteria for IOWA for effectively modeling I/O workloads and seamlessly integrating with arbitrary workload generator and consumer components.

- The I/O workload model should be composed of an ordered set of processes, each identifiable by a unique integer, referred to as a *rank*. Rank values range from 0 to $N-1$, where N is the size of the process group.
- I/O workloads should be described in terms not only of widely available I/O primitives but also of delays between operations and synchronization points across processes.
- Independent streams of I/O operations should be produced for each workload process (such that each process can consume operations at its own rate).
- The ability to "undo" the generation of an operation should be an optional feature of the API to provide compatability with optimistic simulation systems, such as ROSS.

Because of the emergence of numerous high-level I/O libraries (e.g., MPI-IO, HDF5, PnetCDF), many options are available regarding at which layer of the I/O stack to model workloads. We choose to model I/O workloads using POSIX-like file operations because these operations are generally broadly portable across storage system implementations and simulation models. Support for high-level I/O operations could be incorporated into IOWA, but this requires that I/O consumers be able to appropriately reproduce the high-level operation. For example, additional simulation models may be necessary in order to convert high-level I/O operations into the native operations offered by a given storage system model. Similarly, replaying high-level I/O language workloads may require porting the language to a new platform.

The actual file I/O operations currently supported by IOWA include open, close, read, and write. Combining these basic I/O operations with a delay operation for representing application computational phases provides sufficient mechanisms for a workload generator method to model independent I/O workloads. However, an additional mechanism for synchronizing across application processes is needed in order to adequately model parallel I/O workloads. To address this issue, IOWA provides a barrier operation that forces synchronization across all application processes. Consider the behavior of a collective, two-phase I/O [38] operation as an example of how this construct may be used. In the two-phase I/O algorithm, I/O operations are split into separate communication and I/O phases, where all processes communicate with a subset of *aggregator* processes that are responsible for performing the actual file I/O operations (to sequential, non-overlapping portions of the shared file, called *file domains*). The communication step acts as an implicit synchronization point that can be represented by using a barrier operation in IOWA.

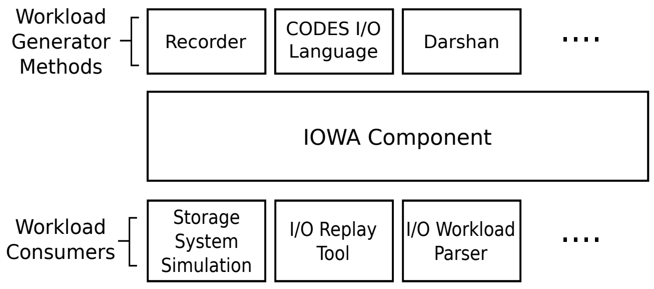

Figure 1: Interaction of IOWA component with arbitrary workload generators and consumers.

In Figure 1, we provide a diagram illustrating the interaction of the IOWA component with example I/O workload generator methods and workload consumers. The respective workload generator methods for Recorder traces, CODES I/O language kernels, and Darshan characterizations are described in detail in the following sections. As part of our evaluation of different workload generator methods, we interfaced multiple workload consumer components with IOWA, including a storage system simulation model, an I/O replay tool, and a text-based parsing tool for debugging purposes.

Workload consumers interface with IOWA through a minimalistic API, composed of only two functions. The `iowa_workload_load` function is responsible for initializing the context necessary to generate workload events (e.g., loading an I/O trace into memory). After the workload has been initialized, events for a given process may be retrieved one at a time by using the `iowa_workload_get_next` function. This function returns a structure identifying the type of workload operation, as well as any respective parameters. The workload generator may be continually polled in this manner until a special event is returned indicating the end of the workload stream.

In order for IOWA to be compatible with optimistic discrete event simulation (the third goal of our design), the API must also offer some support for reverse computation [12]. This implies that each I/O event must be reversible in order to resolve optimistic event timestamp conflicts. The `iowa_workload_get_next` function therefore has a companion function called `iowa_workload_get_next_undo` that will return an I/O operation to the IOWA abstraction layer, in effect allowing consumers to move forward and backward in virtual time. Reversed operations are stored in an in-memory, per-stream queue so that they can be subsequently reissued in the correct order without perturbing the workload generator module. Thus, workload generator modules need not be reverse computation aware themselves.

3.2 Recorder workload generator

As specified in the design requirements of IOWA, we focus strictly on modeling I/O workloads at the POSIX layer for this study. Accordingly, we configured Recorder to trace POSIX I/O functions of interest (e.g., `read` and `write`) to accurately reproduce each application I/O operation. In addition, we configured Recorder to trace MPI-IO functions to help capture synchronization points in the original workload. Tracing both POSIX and MPI-IO functions allows us to correlate which workload I/O operations were issued by high-level collective I/O operations and allows us to model the synchronization inherent in these collective operations.

Since Recorder stores a timestamped record for each application I/O operation, the workload generator method can be implemented just by parsing each record in the trace file and generating corresponding I/O workload operations for functions of interest. The relevant arguments to each traced I/O operation (i.e., the access size and offset) are stored in the trace file, allowing the workload generator to regenerate I/O operations of the target application with total accuracy. To model the computational phases of the target application, the workload generator simply calculates the time deltas between consecutive I/O operations using the timestamps and I/O operation durations logged in the trace file.

3.3 CODES I/O Language Workload Generator

The CODES I/O language was left mostly unchanged from its original implementation [27], save a minor change to allow the language to represent delays with a finer resolution. While the I/O language does not include any specific mechanisms for describing collective I/O operations, it does offer a barrier construct that can be used in conjunction with other independent I/O operations to adequately model collective I/O behavior, as previously mentioned in Section 3.1. The CODES I/O language workload generator method simply uses a parser to translate the given language description into a sequence of I/O operations for each process.

3.4 Darshan Workload Generator

Darshan I/O characterizations maintain detailed records for each file opened by an instrumented application. These file records include counters, timers, and other statistics characterizing the I/O workload imposed by the application. The salient data that Darshan tracks in each file record include the following:

- I/O operation counts for multiple I/O APIs (POSIX, MPI-IO, HDF5, and PnetCDF)
- Timestamps indicating when a file was opened and closed and when the first and last read/write operations occurred
- Cumulative timers indicating how much time was spent performing I/O
- Histograms of I/O access sizes and the four most common individual access sizes

However, accurately reproducing I/O workloads using Darshan logs poses a significant challenge. The compact format used by Darshan to characterize application I/O behavior omits several details that would be helpful in reconstructing the original workload. For example, Darshan records the time span in which I/O occurs and the number of I/O operations, but it does not indicate precisely how those I/O operations are distributed within that time span. For files that are opened collectively, Darshan further collapses all individual file records into an aggregate record for all processes, obscuring the role of individual processes in the collective I/O pattern. To overcome these limitations, we apply basic heuristics to classify the I/O strategy for a given file and then derive representative workload streams based on simplifying assumptions for that classification. This approach

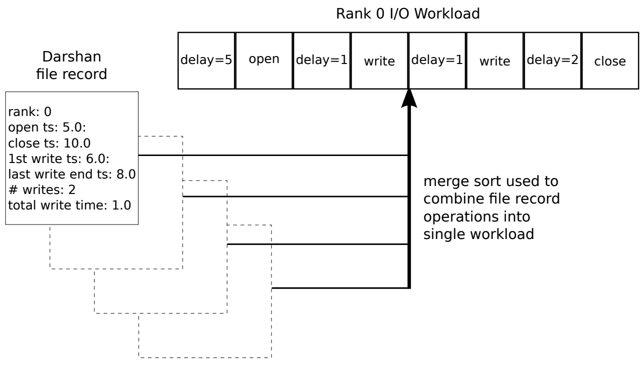

Figure 2: Transforming Darshan file records into an I/O workload.

allows us to reproduce I/O using distinct strategies tailored for certain types of workloads (e.g., independent POSIX vs collective HDF5 workloads).

Figure 2 shows the general process of transforming Darshan I/O characterizations into comprehensive I/O workloads. At a high level, the process involves converting Darshan file records into an ordered (by timestamp) set of I/O events, then merge sorting the events from individual records into a complete workload stream. More specifically, the generator for a specific process iterates the records stored in the Darshan log (sorted according to process rank first, and time of first open second) generating I/O events for each independent file record corresponding to its rank, as well as any shared file records. File read and write operations are evenly distributed across the time span in which Darshan observed I/O activity, as illustrated in Figure 2. A uniform distribution based on the total idle time observed by Darshan (i.e., the duration between initial and final I/O operations minus the cumulative I/O time for the file) is used to model the computational delays between successive I/O operations. This approach allows the regenerated workload to achieve a similar aggregate I/O rate to the file, although it will not capture uneven bursts of activity in the I/O stream. We leave exploration of alternative distribution functions (e.g. Poisson or Pareto) for future work. It is not yet clear if such distributions would be effective at modeling the I/O request rate for individual processes.

Regenerating I/O workloads from Darshan shared file records is more complicated, since important data regarding the I/O workload is lost when the individual file records are condensed into a single shared record. We therefore use heuristics to classify the I/O patterns of these shared file records into two cases: independent I/O to a shared file and collective I/O to a shared file. If the shared file record counters indicate that high-level (e.g., MPI or HDF5) collective I/O operations were issued to the file, then the generator will use synchronization operations to emulate the collective I/O algorithm. In the independent I/O case, I/O operations are assigned to all workload processes in round-robin fashion until no operations are left to generate. In the collective I/O case, we emulate the behavior of two-phase I/O: a subset of workload processes is selected to be responsible for performing I/O to the file (i.e., aggregator processes), and then I/O operations are similarly assigned in round-robin fashion across this subset of processes. In each case, we assume that each set of processes submitting I/O operations to the file is responsible for its own independent file domain.

The access size for each operation is chosen based on histograms of access sizes as well as the most common access sizes observed by Darshan. For simplicity, offsets are assigned sequentially in a file.

4. EXAMPLE USE CASES

In this section we examine the behavior of each of the three workload generators for example I/O workload use cases. All application and replay examples were executed on Mira, an IBM Blue Gene/Q supercomputer maintained by the Argonne Leadership Computing Facility (ALCF). Mira consists of 786,432 cores and 768 terabytes of memory, capable of achieving a peak performance of about 10 petaflops. More relevant to this study is the high-performance I/O subsystem available to Mira users—a GPFS file system offering 24 petabytes of capacity and up to 240 GiB/second bandwidth.

4.1 Storage system simulation

One of the most straightforward applications of IOWA is to generate representative workloads for storage system simulations. This capability can be used to ensure that storage architecture and algorithm models are evaluated by using relevant workloads. To test this functionality, we integrated IOWA into an existing discrete-event storage system model based on Intrepid, a recently decommissioned IBM Blue Gene/P system. This storage system model has been validated in previous work [26] and used to explore the impact of burst buffers in HPC storage architectures [27]. The Intrepid model already included each major component of the IBM Blue Gene/P storage architecture, including storage devices, file servers, I/O forwarding nodes, compute nodes, and interconnects. The compute node entities were updated to ingest operations from IOWA in order to inject I/O events into the storage system.

We model the I/O pattern of VPIC-IO as our initial simulation test case. VPIC-IO is an I/O kernel of the VPIC plasma physics simulation code developed by Los Alamos National Laboratory [4]. VPIC-IO leverages H5Part [2], an API that offers a high-performance parallel data interface to HDF5 intended for storing time-varying datasets in particle physics applications. VPIC-IO writes a 1D particle array into a shared HDF5 file, with each particle containing eight distinct variables. We analyzed a representative VPIC configuration with 8,192 MPI processes (16 MPI processes per compute node) in order to focus on a specific scenario with detailed simulator instrumentation. The Recorder traces and Darshan logs were obtained by using link-time instrumentation on Mira, while the CODES I/O language description of VPIC-IO was crafted by hand. The I/O language representation for VPIC-IO was particulary cumbersome to generate, since it was responsible for emulating the two-phase I/O algorithm used to implement collective I/O operations.

Because the Intrepid system has been decommissioned, we were not able to directly validate simulation behavior against a real-world system. We can contrast the three workload generators in simulation, however. Figure 3 plots the aggregate number of write operations observed over 40 distinct intervals in the simulation of each workload generator's representation of the VPIC-IO workload. While the write rates of each generator appear similar, subtle differences ex-

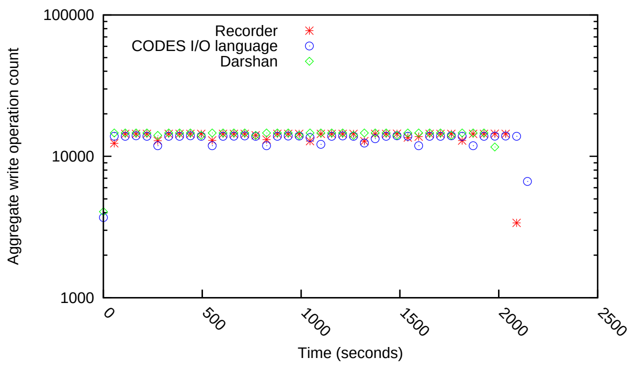

Figure 3: Aggregate write I/O operation counts over 40 distinct intervals for each workload generator method when simulating the VPIC I/O workload using the CODES Intrepid storage system model.

ist between the generated workloads causing the CODES I/O language and Recorder workloads to experience 8% and 5% increases in execution time, respectively. The increases are due largely to periodic dips in write request rates at the end of each collective write operation that appear in the CODES I/O language and Recorder workload generator methods but not in the Darshan method.

For the CODES I/O language workload generator, this reduced write rate is an artifact of how the composed I/O descriptions model the two-phase I/O algorithm. While the CODES I/O language representation of the VPIC-IO workload correctly reproduces the aggregate write volume of the collective I/O operations, the lack of expressiveness in the I/O language results in an uneven distribution of this workload among aggregator processes. Specifically, the I/O language representation assumes that all I/O operations of a collective are the same size (i.e., the size of the collective I/O buffer), rather than using smaller I/O sizes when possible. Hence, in the last round[1] of collective I/O, only a subset of aggregators may be performing I/O operations, rather than breaking the operations into smaller chunks such that no aggregators are idle.

The reason for the reduced write rate in the Recorder workload generator is related to how this method models delays between consecutive operations. Because Recorder is based on a direct trace of the application's I/O behavior, it generates delays verbatim as desribed in the original trace. Thus, it may inadvertently reproduce runtime anomalies, such as straggling processes, that reduce the overall I/O rate. The Darshan and CODES I/O language generators avoid this issue by distributing delays heuristically to average out any variations in delay between I/O operations.

4.2 I/O workload replay

Another application of the IOWA workload model is for replay of representative workloads on actual HPC systems. This capability can be used to evaluate application perfor-

[1]A collective I/O operation is composed of potentially numerous *rounds*, where a round is one repetition of the two-phase algorithm (i.e., a communication phase and an I/O phase). The number of rounds depends on the aggregate I/O size, the collective I/O buffer size, and the available number of aggregators.

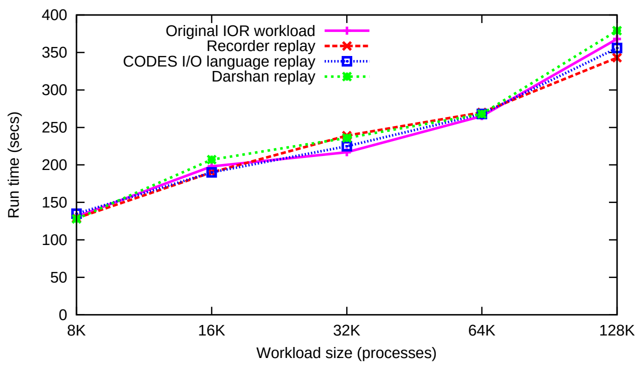

Figure 4: Measured runtime for original IOR file-per-process workload and each IOWA generated workload replay on Mira.

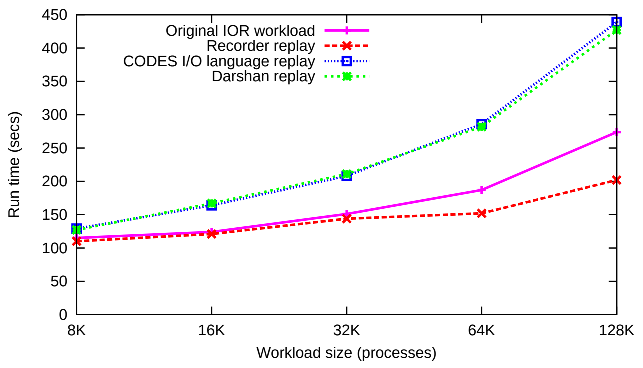

Figure 5: Measured runtime for original IOR shared file workload and each IOWA generated workload replay on Mira.

mance on new platforms or to investigate I/O tuning parameters using a proxy application that is simpler to configure and execute than the original scientific application. As mentioned previously, we have developed a generic MPI-based replay tool that interfaces with IOWA to regenerate workloads on real HPC systems. This replay tool uses POSIX calls to replay I/O operations, MPI barrier operations to replay synchronization, and a sleep mechanism to replay application computation.

We elected to use the IOR benchmark as a simple initial test case for this functionality. IOR is the de facto standard I/O benchmark for HPC storage systems and can be configured to generate a wide variety of I/O workloads [35]. We configured IOR to use a unique file on each process, with each process writing 64 MiB of data using 4 MiB POSIX write operations. Again, Recorder traces and Darshan logs for IOR were obtained by executing IOR profiling jobs with link-time instrumentation on Mira. The CODES I/O language description of IOR was created by hand according to the selected IOR configuration. Note that we configured the replay tool to ignore the delay operations in this experiment, since the amount of time IOR spent computing is essentially negligible compared with the time spent doing I/O.

Figure 4 compares the runtime of the original IOR benchmark with the runtime of the replay tool using the three example workload generators as we vary the scale from 8,192 MPI processes to 131,072 MPI processes. All IOR and I/O replay executions used 32 MPI processes per compute node. One can see that each of the workload generators tracks closely to the performance of the original IOR workload, with no more than 10% error at any scale. This error is within the anticipated amount of I/O performance variance on this platform due to contention and other external factors. This workload is straightforward for each generator to model because it does not involve any collective I/O or computation.

5. I/O MODELING CHALLENGES

In the previous section we demonstrated that IOWA can be used to drive arbitrary workload consumers, such as storage system simulation models and I/O replay tools, using I/O workload data from a variety of sources. During the course of our experiments, we encountered numerous challenges related to I/O modeling accuracy, however. We further observed that some of these challenges are subtle and therefore have the potential to mask themselves from researchers attempting to generate representative models of I/O workloads for a target system. In this section we investigate some of these key challenges and propose best practices for addressing them.

5.1 Collective operations

Section 4.2 demonstrated the use of IOWA for replay of an IOR workload with each process performing independent POSIX I/O to separate files. We next modified the IOR configuration to use MPI-IO rather than POSIX for I/O, use an interleaved I/O pattern, and enable collective I/O. Although this example writes the same amount of data, it uses two-phase I/O in ROMIO [38] to redistribute data to a subset of processes (known as aggregators) before writing data to the storage system. The results of this experiment are plotted in Figure 5. This example shows a greater disparity between the original workload execution time and the I/O replay execution times than we observed in the file-per-process example. At modest scales (up to 32K processes), the Recorder workload generator attains comparable performance (no more than 5% error) to that of the original application. At larger scales, the performance begins to diverge slightly, ultimately exhibiting a roughly 26% decrease in runtime compared with the target workload at 128K processes. The I/O language and Darshan workload generators exhibit much worse performance than the target workload and diverge more rapidly than than the Recorder workload generator, resulting in an increase in runtime of over 55% in each case at a scale of 128K processes. We determined these pronounced performance disparities to be related to subtle issues in the manner in which each workload generator models collective I/O behavior.

Specifically, the CODES I/O language and Darshan workload generators fail to match the performance of the original application because they do not take into account platform-specific topology information. The Blue Gene/Q MPI-IO implementation on Mira[2] contains platform-specific optimizations to select optimal aggregator processes as well as optimal file domains for each aggregator. In particular, the aggregator selection algorithm favors *bridge nodes* that can

[2]`https://repo.anl-external.org/repos/bgq-driver/V1R2M2/bgq-V1R2M2.tar.gz`

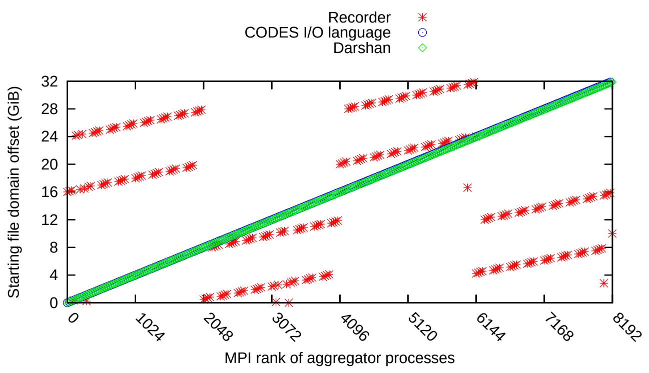

Figure 6: File domain offsets for aggregators in each IOWA workload generator's model of an IOR collective workload. Note that the aggregator and file domain assignments in the CODES I/O language and Darshan workloads are nearly identical, causing occlusion of the CODES I/O language points.

achieve higher throughput to the storage system, and file domains are chosen to reduce lock contention by aligning to the GPFS block size. The Recorder workload generator is not susceptible to this problem because it reproduces the precise aggregators, offsets, and sizes from the original application. We illustrate the problem in greater detail in Figure 6, which shows the starting offset (i.e., file domain) for each aggregator process in a 8,192 process run of the IOR shared file workload. The Darshan and CODES I/O language generators assign file domains sequentially throughout the file in a similar manner. However, the Recorder generator reveals that the aggregators are actually assigned in an unexpected manner, likely related to the layout of MPI processes and the topology of compute nodes on Mira. This discrepancy did not affect our simulation results (since the model does not reflect Mira's topology), but it is clearly an issue for workload replay on a real system.

The Recorder workload generator, in contrast, produces a workload stream that exceeds the performance of the original application at larger scales. We believe that this is due to the fact that IOWA's POSIX-level I/O representation does not allow us to capture the communication costs associated with collective operations at scale. Instead, the communication costs of two-phase I/O are modeled only by a barrier operation. When I/O time is the dominant factor in performance, this approach is adequate; but when network transfer times are significant (e.g., in the presence of contention), this modeling strategy falls short.

These results suggest the need for more accurate models of high-level I/O libraries and, in particular, collective I/O operations. This is a relatively simple problem to address when using IOWA to replay I/O workloads on a real system, assuming the necessary libraries are available for replaying the desired I/O operations: new operations can be added to IOWA to model the high-level I/O operations, and the replay tool can be linked with the necessary library to replay these operations. This would allow IOWA to transparently model the underlying implementations of these I/O operations with minimal effort and greater accuracy. This strategy would be difficult to adopt in simulations studies, however, as it would require the simulator to faithfully model

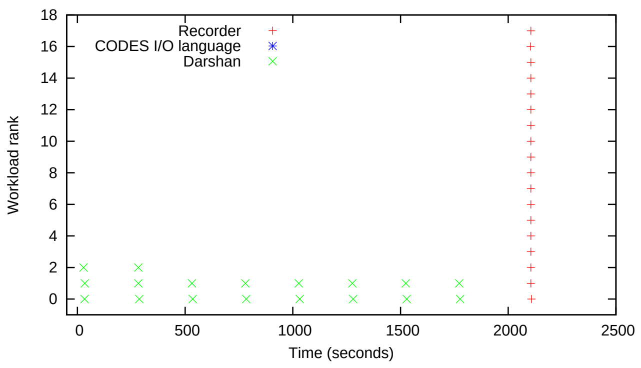

Figure 7: Distribution of HDF5 metadata writes in simulation of the VPIC-IO workload using the CODES Intrepid storage system model.

both the I/O libraries and any topology-aware optimizations that they might contain.

Best practices: High-fidelity modeling of collective I/O algorithms generally requires accounting for platform-specific optimizations and topology details. I/O workload generators and consumers should capture and replay high-level I/O operations directly or else take care to ensure that the workload data sources include enough topology and placement information for authentic workload recreation.

5.2 High-level I/O library metadata

Another challenge related to accurately modeling I/O workloads is reproducing subtle, internal I/O traffic produced by high-level I/O libraries. For instance, a high-level I/O library may maintain a header or other metadata for a given dataset that is concealed from users. While these operations may represent only a negligible fraction of the application's aggregate I/O, the manner in which they are reproduced has the potential to perturb the observed I/O performance of the workload.

To illustrate this problem, we further analyze the VPIC-IO benchmark from Section 4.1, again using the CODES Intrepid storage model. The VPIC-IO benchmark leverages the HDF5 I/O library, which stores internal metadata to assist in maintaining the time-varying particle datasets using the HDF5 file format. Figure 7 indicates how these metadata operations are regenerated (i.e., what time and by which application process) using each workload generator's representation of the VPIC-IO workload. The larger, bulk data operations are omitted from this graph for clarity.

Note that we do not illustrate how the CODES I/O language workload generator handles this issue. Since the I/O kernels are developed manually, the manner in which these metadata operations are reproduced is dictated by the kernel developer and is thus uninteresting to analyze further. To accurately reproduce this type of workload, an I/O kernel developer would likely need assistance from other I/O analysis tools (e.g., a tracing tool) or an in-depth knowledge of the high-level I/O library implementation.

Since the Recorder traces at the POSIX layer (i.e., below the HDF5 layer), the Recorder workload generator can reproduce the HDF5 metadata operations with total accuracy. The metadata operations are actually submitted to the storage system at the end of the I/O workload, after the eight

Table 1: Original IOR file-per-process workload runtime with varying memory allocation strategies.

Allocation Strategy	Average Runtime (sec.)
default	270.7
dynamic uninitialized	340.2
pre-open	224.9

VPIC particle variables have been collectively written to file. Specifically, application process ranks 0–17 each perform a single independent write before closing the HDF5 file. The Darshan workload generator correctly detects the presence of the independent writes in the target workload, but it does not distribute them correctly over time or processes because the Darshan characterization lacks the level of detail needed to do so. One potential way to help reproduce this type of workload more accurately would be to modify Darshan I/O characterizations to store timing information separately for collective I/O operations and independent I/O operations.

Best practices: Application I/O workload models should encompass indirect traffic produced by high-level library data structures.

5.3 Memory allocation

Another key challenge that we encountered in I/O workload modeling was the unexpected impact of memory allocation strategies on overall I/O performance of our replay tool. The replay tool prototype initially produced poor agreement with the I/O performance of the original application runs regardless of the generator being used. The root cause was determined to be a subtle difference in memory buffer allocation strategies between the original application (IOR in this case) and the replay tool.

We investigated this issue using the original IOR benchmark as our test case in order to eliminate the workload replay tool itself as a potential source of noise. Table 1 shows the difference in average IOR execution time on Mira for a fixed configuration as we modify the memory allocation strategy in IOR. We collected 10 independent samples for each configuration in order to mitigate the impact of system noise on the results. The three test cases can be described as follows:

- *Default*: IOR's default configuration: the memory buffer to be used for I/O is allocated after creating the output file. All bytes of the buffer are also initialized before the first I/O operation occurs.
- *Dynamic uninitialized*: A modified version of IOR in which a new buffer is allocated (and subsequently freed) for each I/O operation. The memory is not initialized before it is written.
- *Pre-open*: Same as the *default* case except that the buffer is allocated and initialized before the file is created rather than after it is created.

The *dynamic uninitialized* reflects the behavior of the initial replay tool prototype, but it clearly exhibits reduced performance compared with that of the original IOR application. We elected to use the *default* approach in the replay tool in order to be consistent with the strategy used by IOR. In practice, a real application may be more likely to match the *pre-open* behavior, however. This case reflects the standard practice of completing a round of computation (with results in memory) prior to opening an output file to write checkpoint or visualization data. Unfortunately, it is not clear how to reliably determine which method an application is using. The performance difference between these approaches is surprising (over 100 seconds of runtime) and warrants further investigation into the Blue Gene compute node kernel implementation. We believe that this phenomenon not only would affect replay tools but also would perturb synthetic benchmarks.

Best practices: Faithful recreation of I/O access patterns is not necessarily sufficient to ensure accurate replay; memory allocation strategies also play a key role in I/O performance.

6. RELATED WORK

6.1 I/O tracing tools

Several tracing tools have been developed to address different HPC I/O workload analysis challenges. These tools are complementary to IOWA in that they could potentially be integrated as generator modules. //TRACE [31] is an I/O trace replay tool that automatically discovers inter-node dependencies and inter-I/O arrival rates to accurately recreate application I/O behavior. ScalaIOTrace [43] is a scalable, multilevel MPI-IO tracing framework that includes a trace replay engine; it can optionally replay MPI communication workloads as well. IPM [39] is a framework for collecting, profiling, and aggregating HPC performance information, including data on the performance of parallel I/O operations. TBBT [44] is an NFS trace replay tool that supports spatial and temporal scaling of trace workloads. HDTrace [22] is a framework for tracing MPI programs and replaying these traces on either real or simulated clusters. HDTrace also traces PVFS client and server activity and gathers operating system, network, I/O, and CPU statistics to enable correlation across the software stack. The traces can be ingested by PIOsimHD, a discrete event simulator.

6.2 Synthesizing HPC I/O workloads

Synthesizing I/O workloads is an attractive alternative to full I/O tracing because it addresses many of the shortcomings of traces (e.g., lessened storage requirements, flexibility to modify specific workload parrameters). In general, synthesizing representative I/O workloads requires some sort of model or characterization of the target application's I/O behavior. In most existing research, HPC application I/O characterizations are generated by first tracing the I/O operations of a target application and performing an in-depth analysis of this trace offline [5, 14, 18, 30, 37]. In contrast, Darshan I/O characterizations are automatically generated at runtime.

Some novel workload generation techniques also have been proposed in the literature, which could be applied in the context of HPC I/O workloads. Kao presents a workload generator technique that allows for generating workloads according to numerous user "populations," each of which can be configured independently according to user-supplied distributions [20]. This functionality lends itself to the generation of ensemble HPC I/O workloads, where numerous users with distinct I/O requirements compete for access to a shared file system. Another interesting approach is given by Kurmas et al., where workloads are automatically generated by iteratively distilling the workload parameters that have the greatest impact on I/O performance out of some target workload [23]. This approach is interesting because it can yield representative I/O workloads for a given application with no human intervention. He et al. present PIONEER [18], a parallel I/O workload characterization and generation framework, which attempts to address open problems in parallel I/O workload modeling, such as interprocess correlations and I/O library request dependencies. The PIONEER approach is to analyze a workload's trace files offline in order to determine interprocess correlations and to create a generic workload presentation that is used by all workload processes to regenerate the workload. Also, knowledge of I/O library request dependencies is used to enforce a sensible ordering of workload operations.

6.3 Parallel file system simulations

Several parallel file system simulators (examples of potential consumers of IOWA workload models) have been proposed in the literature. The design philosophy of IMPIOUS [32] centers on the use of simple, abstract file system component models that enable easy adaptation. PFSsim [28] and SIMCAN [33] are two other highly modular and configurable parallel file system simulators based on the OMNeT++ [41] network simulation framework. PFSsim is designed specifically for the efficient evaluation of different I/O scheduling algorithms, whereas SIMCAN emphasizes easily simulating a range of HPC architectures and application I/O patterns. FileSim [16] is another parallel file system simulator that is geared toward end-to-end I/O performance prediction and analysis of exascale HPC systems. FileSim's parallel simulation framework has been demonstrated using large-scale file system models containing up to 52,000 clients [16].

Each of these simulators is driven primarily by I/O traces, although some frameworks do provide support for generating synthetic I/O workloads as well. IMPIOUS provides users with generic I/O workload generators that can produce both file-per-process and shared file checkpointing workloads, common among most HPC applications. SIMCAN provides mechanisms for modeling HPC applications using state graphs, which can in turn be used to drive the simulation and analysis of the storage system model.

7. CONCLUSIONS

In this work we demonstrated the design of IOWA, a novel workload abstraction layer that may be used by diverse tools to regenerate and analyze I/O workloads. We implemented three workload generators based on distinct representations of I/O workloads: I/O traces, synthetic I/O kernels, and I/O characterizations. We used a simulation model of an HPC storage system to analyze and compare each IOWA workload generation technique in detail, and we used an I/O replay engine to evaluate each generator's accuracy in regenerating large-scale workloads on a production HPC storage system. We also presented a set of best practices for practitioners interested in generating HPC I/O workloads based on some of the challenges we encountered during this research.

We found that each workload generation method offers its own inherent tradeoffs related to accuracy, flexibility, and breadth of use. The Recorder generator consistently reproduces the target workloads with the most accuracy, making it the best workload generation method for in-depth study of a specific application workload of interest. The I/O language and Darshan workload generators also perform well for independent workloads, but they lack the necessary detail to reproduce collective I/O workloads as accurately as a high-resolution trace replay.

Table 2: Size (in KiB) of source workload files for each application example from this study (each using 8K workload processes).

Workload Source	I/O Workload		
	IOR file per process	*IOR shared file*	*VPIC-IO*
Recorder (compressed)	3,745.77	2,838.97	3,921.56
CODES I/O Language	.72	13.71	27.44
Darshan	1,391.97	.65	.65

Of our proposed workload generation methods, the CODES I/O language is the most suited for studying hypothetical or projected workloads, since the generated workload pattern is crafted manually and conducive to parameterization. It is by far the most labor-intensive method used in this study, however, especially when the goal is to recreate the access pattern of a specific target application.

Darshan's lightweight nature has led it to be enabled by default on a number of production HPC storage systems, providing researchers access to a broad collection of representative HPC I/O workloads. The Darshan logs are also the most conducive to collaboration because of their small size and Darshan's anonymization capability. We have demonstrated the initial design of a workload generation technique that can produce reasonable workloads from these Darshan I/O characterizations, enabling researchers to easily use these workloads in future studies.

Table 2 provides data on the size of the workload representations for each workload generator for each application workload evaluated in this study. Clearly, the CODES I/O language and Darshan datasets are typically smaller than the Recorder traces by orders of magnitude. Note that we give the size of the compressed Recorder traces even though they must be uncompressed before consumed by the Recorder workload generator method.

To summarize, we have found that each of our proposed workload generation techinques is amenable to specific use cases; determining which generator to use depends on the type of I/O analysis to be performed. We have also demonstrated that modeling I/O workloads at the lowest level of the I/O stack (i.e., POSIX-level) is the most generally portable option, but likely at the cost of accuracy in modeling high-level I/O workloads, such as collective workloads. In future work, we hope to integrate more workload generation techniques into IOWA, including generators utilizing probability distributions and generators that can extrapolate a given workload to larger scale. We also hope to refine our technique for synthesizing workloads from Darshan characterizations by gathering more detailed information on application I/O workloads and leveraging this data to increase workload regeneration accuracy.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific

Computer Research, under contract DE-AC02-06CH11357. The research used resources from Argonne Leadership Computing Facility. This work is also supported in part by the Director, Office of Laboratory Policy and Infrastructure Management of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

8. REFERENCES

[1] mdtest benchmark. `http://sourceforge.net/projects/mdtest/`, 2015.

[2] A. Adelmann, R. Ryne, J. Shalf, and C. Siegerist. H5Part: A portable high performance parallel data interface for particle simulations. In *Particle Accelerator Conference, 2005. PAC 2005. Proceedings of the*, pages 4129–4131. IEEE, 2005.

[3] D. W. Bauer Jr, C. D. Carothers, and A. Holder. Scalable time warp on blue gene supercomputers. In *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, pages 35–44. IEEE Computer Society, 2009.

[4] K. J. Bowers, B. Albright, L. Yin, B. Bergen, and T. Kwan. Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation. *Physics of Plasmas (1994-present)*, 15(5):055703, 2008.

[5] S. Byna, Y. Chen, X.-H. Sun, R. Thakur, and W. Gropp. Parallel I/O prefetching using MPI file caching and I/O signatures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 44. IEEE Press, 2008.

[6] D. Capps and W. Norcott. IOzone filesystem benchmark. `http://www.iozone.org/`.

[7] P. Carns. ALCF I/O data repository. Technical Report ANL/ALCF/TM-13/1, Argonne National Laboratory (ANL), 2013.

[8] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross. Understanding and improving computational science storage access through continuous characterization. *ACM Transactions on Storage (TOS)*, 7(3):8, 2011.

[9] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley. 24/7 characterization of petascale I/O workloads. In *Proceedings of 2009 Workshop on Interfaces and Architectures for Scientific Data Storage*, September 2009.

[10] P. Carns, Y. Yao, K. Harms, R. Latham, R. B. Ross, and K. Antypas. Production I/O characterization on the Cray XE6. In *In Proceedings of the Cray User Group meeting 2013 (CUG 2013)*, 2013.

[11] C. D. Carothers, D. Bauer, and S. Pearce. Ross: A high-performance, low-memory, modular time warp system. *Journal of Parallel and Distributed Computing*, 62(11):1648–1669, 2002.

[12] C. D. Carothers, K. S. Perumalla, and R. M. Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 9(3):224–253, 1999.

[13] J. Cope, N. Liu, S. Lang, P. Carns, C. Carothers, and R. Ross. Codes: Enabling co-design of multilayer exascale storage architectures. In *Proceedings of the Workshop on Emerging Supercomputing Technologies*, 2011.

[14] P. E. Crandall, R. A. Aydt, A. A. Chien, and D. A. Reed. Input/output characteristics of scalable parallel applications. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, page 59. ACM, 1995.

[15] Department of Energy. CORAL. `http://asc.llnl.gov/CORAL-benchmarks/`, 2015.

[16] S. Eidenbenz, M. Erazo, T. Li, and J. Liu. Toward comprehensive and accurate simulation performance prediction of parallel file systems. Technical report, Los Alamos National Laboratory (LANL), 2011.

[17] S. Godard. Sysstat utilities home page. `http://sebastien.godard.pagesperso-orange.fr/`, 2015.

[18] W. He, D. H. Du, and S. B. Narasimhamurthy. PIONEER: A solution to parallel I/O workload characterization and generation. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 111–120. IEEE, 2015.

[19] M. Heroux and R. Barrett. Mantevo project. `https://mantevo.org/`, 2015.

[20] W.-I. Kao and R. K. Iyer. A user-oriented synthetic workload generator. In *Distributed Computing Systems, 1992., Proceedings of the 12th International Conference on*, pages 270–277. IEEE, 1992.

[21] Y. Kim, R. Gunasekaran, G. M. Shipman, D. A. Dillow, Z. Zhang, and B. W. Settlemyer. Workload characterization of a leadership class storage cluster. In *5th Petascale Data Storage Workshop (PDSW)*, pages 1–5. IEEE, 2010.

[22] J. Kunkel. HDTrace – a tracing and simulation environment of application and system interaction. *Hamburg. University of Hamburg–2011*, 2011.

[23] Z. Kurmas, K. Keeton, and K. Mackenzie. Synthesizing representative I/O workloads using iterative distillation. In *Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003. 11th IEEE/ACM International Symposium on*, pages 6–15. IEEE, 2003.

[24] Lawrence Livermore National Laboratory. IOR benchmark. `https://github.com/chaos/ior`, 2015.

[25] Lawrence Livermore National Laboratory. Lustre Monitoring Tool (Github). `https://github.com/chaos/lmt`, 2015.

[26] N. Liu, C. Carothers, J. Cope, P. Carns, R. Ross, A. Crume, and C. Maltzahn. Modeling a leadership-scale storage system. In *Parallel Processing and Applied Mathematics*, pages 10–19. Springer, 2012.

[27] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn. On the role of burst buffers in leadership-class storage systems. In *Proceedings of 28th IEEE MSST conference*, 2012.

[28] Y. Liu, R. Figueiredo, D. Clavijo, Y. Xu, and M. Zhao. Towards simulation of parallel file system scheduling algorithms with PFSsim. In *Proceedings of the 7th IEEE International Workshop on Storage Network Architectures and Parallel I/O (May 2011)*, 2011.

[29] H. Luu, B. Behzad, R. Aydt, and M. Winslett. A multi-level approach for understanding I/O activity in HPC applications. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–5, Sept 2013.

[30] S. Méndez, D. Rexachs, and E. Luque. Modeling parallel scientific applications through their input/output phases. In *Cluster Computing Workshops (CLUSTER WORKSHOPS), 2012 IEEE International Conference on*, pages 7–15. IEEE, 2012.

[31] M. P. Mesnier, M. Wachs, R. R. Sambasivan, J. Lopez, J. Hendricks, G. R. Ganger, and D. O'Hallaron. Trace: Parallel trace replay with approximate causal events. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, pages 24–24, Berkeley, CA, USA, 2007. USENIX Association.

[32] E. Molina-Estolano, C. Maltzahn, J. Bent, and S. Brandt. Building a parallel file system simulator. In *Journal of Physics: Conference Series*, volume 180, page 012050. IOP Publishing, 2009.

[33] A. Núñez, J. Fernández, J. D. Garcia, F. Garcia, and J. Carretero. New techniques for simulating high performance MPI applications on large storage networks. *The Journal of Supercomputing*, 51(1):40–57, 2010.

[34] P. C. Roth. Characterizing the I/O behavior of scientific applications on the Cray XT. In *Proceedings of the 2nd International Workshop on Petascale Data Storage*, pages 50–55, New York, NY, USA, 2007. ACM.

[35] H. Shan, K. Antypas, and J. Shalf. Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 42. IEEE Press, 2008.

[36] S. S. Shende and A. D. Malony. The tau parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[37] E. Smirni and D. A. Reed. Workload characterization of input/output intensive parallel applications. In *Computer Performance Evaluation Modelling Techniques and Tools*, pages 169–180. Springer, 1997.

[38] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective I/O in ROMIO. In *The Seventh Symposium on the Frontiers of Massively Parallel Computation, 1999. Frontiers' 99.*, pages 182–189. IEEE, 1999.

[39] A. Uselton, M. Howison, N. J. Wright, D. Skinner, N. Keen, J. Shalf, K. L. Karavanic, and L. Oliker. Parallel I/O performance: From events to ensembles. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–11. IEEE, 2010.

[40] R. F. Van der Wijngaart and P. Wong. NAS parallel benchmarks version 2.4. Technical report, NAS technical report, NAS-02-007, 2002.

[41] A. Varga et al. The OMNeT++ discrete event simulation system. In *Proceedings of the European Simulation Multiconference (ESMâĂŹ2001)*, 2001.

[42] J. Vetter and C. Chambreau. mpiP: Lightweight, scalable MPI profiling. 2014.

[43] K. Vijayakumar, F. Mueller, X. Ma, and P. C. Roth. Scalable I/O tracing and analysis. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, pages 26–31, New York, NY, USA, 2009. ACM.

[44] N. Zhu, J. Chen, T.-C. Chiueh, and D. Ellard. TBBT: scalable and accurate trace replay for file server evaluation. In *ACM SIGMETRICS Performance Evaluation Review*, volume 33, pages 392–393. ACM, 2005.