# ScalaIOExtrap: Elastic I/O Tracing and Extrapolation

Xiaoqing Luo, Frank Mueller
Department of Computer Science
North Carolina State University
Raleigh, NC, USA
Email: mueller@cs.ncsu.edu

Philip Carns, Jonathan Jenkins, Robert Latham, Robert Ross, Shane Snyder
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
Email: carns,jenkins,robl,rross,ssnyder@mcs.anl.gov

***Abstract*—Today's rapid development of supercomputers has caused I/O performance to become a major performance bottleneck for many scientific applications. Trace analysis tools have thus become vital for diagnosing root causes of I/O problems. This work contributes an I/O tracing framework with (a) techniques to gather a set of lossless, elastic I/O trace files for small number of nodes, (b) a mathematical model to analyze trace data and extrapolate it to larger number of nodes, and (c) a replay engine for the extrapolated trace file to verify its accuracy. The traces can in principle be extrapolated even beyond the scale of present-day systems and provide a test if applications scale in terms of I/O. We conducted our experiments on three platforms: a commodity Linux cluster, an IBM BG/Q system, and a discrete event simulation of an IBM BG/P system. We investigate a combination of synthetic benchmarks on all platforms as well as a production scientific application on the BG/Q system. The extrapolated I/O trace replays closely resemble the I/O behavior of equivalent applications in all cases.**

## I. Introduction

I/O behavior is one of the key factors that impacts application performance, particularly for large-scale high-performance computing (HPC) and big data analytic applications that rely on parallel file systems (PFSs). I/O presents a challenge due to complex interactions of multiple software components [8]. It is thus imperative to understand inefficiencies and determine bottlenecks in I/O, which is facilitated by tracing and analyzing I/O performance of parallel applications. However, I/O analysis in parallel systems is non-trivial due to multiple I/O layers [21] and multiple I/O patterns. The following general I/O patterns can be distinguished (processors are synonymous for compute tasks on nodes): **(A) Serial I/O (SIO):** Data is aggregated from all the processors to a single processor, the "spokesperson"/proxy, and only the spokesperson performs I/O (PFS). **(B) Parallel I/O, one file per process (N-to-N):** All processors perform I/O simultaneously on individual files (local or PFS), each with a different name/path. **(C) Parallel I/O, shared-file (N-to-1):** Processors perform I/O on a single shared file simultaneously, each within a disjoint block of the file (PFS).

To understanding I/O behavior, two general types of techniques may be employed:

• **Dynamic I/O analysis**, such as ScalaIOTrace [17], [24], which needs to be linked to the original applications and run together with the applications on high-performance computing (HPC) systems. Detailed I/O access information can be collected with such a tracing tool. However, the system overhead of such a tracing tool is significant, especially for a large-scale production HPC system [23] (e.g., long application execution time and large number of nodes participating).

• **Static I/O analysis:** Gather the trace information at compile time. Although such analysis can be performed without actually executing the programs, it requires the access to program sources, which may not be available for some applications. It may also fail to capture I/O patterns that are dependent upon runtime calculations.

I/O tracing can also be performed by modeling and predicting applications' behavior [7]. Unfortunately, such an approach can only provide overall statistics for an application on a particular architecture, and may not satisfy the needs for detailed analysis.

Due to the restrictions of analysis methods mentioned above, we created a novel tool, ScalaIOExtrap. It obtains the lossless I/O access behavior of an application running in a large-scale system without requiring source code. Fig. 1 gives an overview of ScalaIOExtrap, where RS (Rank Size) defines the number of ranks of a job's communicator obtained from MPI_Comm_size.

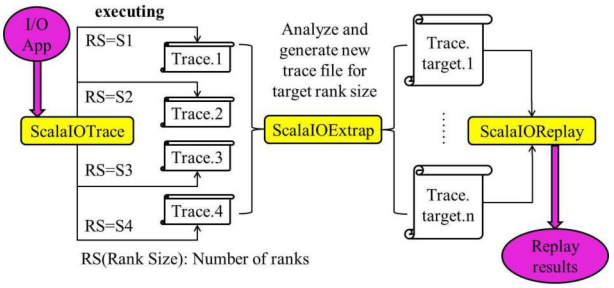


Fig. 1: ScalaIO Framework: Trace, Extrap, Replay

The high level methodology is (1) to gather a set of lossless and scalable I/O trace files in a relatively small system via ScalaIOTrace; (2) to analyze the set of trace files and extrapolate small files into large size trace files via ScalaIOExtrap; (3) to calculate the extrapolated data and generate a single trace file; and (4) to enable I/O replay and verify the correctness of extrapolation via ScalaIOReplay.

For verifying the accuracy and portability of our approach, we conducted experiments on three different HPC platforms in terms of cluster types: (1) a relatively small-scale Linux cluster, (2) a large-scale IBM BG/Q system, (3) and a simulated system in the CODES simulation toolkit [10]. We discuss each of them in Section IV. The results indicate that the extrapolated trace file captured exactly the same behavior as performed by the I/O application. Also, ScalaIOExtrap does not introduce extra overhead to the system, and does not extend the execution time of the applications, as we only use one processor to analyze and compute the I/O parameters. For

example, for the trace information collected from the IOR pseudo application run with 8192 processes, it only takes less than one second for a single processor to generate the extrapolated trace file. In contrast, traditional dynamic I/O analysis tools (e.g., ScalaIOTrace) require 8192 processors with an entire application run plus more than 200 seconds at finalization time to gather the equivalent trace information during inter-node compression (see next section).

## II. Background

ScalaIOTrace and its extrapolation engine are based on related work on MPI tracing via ScalaTrace V2 [24], [23] of which we obtained a copy. In this section, we briefly introduce these tools and CODES, which is utilized to verify our work.
ScalaTrace is an MPI communication tracing framework for parallel applications [16]. It utilizes the MPI profiling layer (PMPI) to intercept MPI calls. ScalaTrace collects lossless, order-preserving, and space-efficient communication traces by exploiting the program structure and performing a two-stage trace compression, i.e., intra-node and inter-node compression while preserving timing [18].
Intra-node compression captures repetitive MPI events in a loop using regular section descriptors (RSDs) as a tuple $\{length, event_1, ...event_n\}$ in constant size [17]. Nested loops become power-RSDs (PRSDs), i.e., recursively structured RSDs. Consider the following MPI-IO example:

```
for( i = 0; i < 10; i++ ) {
        MPI_File_open(...);
    for( j = 0; j < 100; j++) {
        MPI_File_write(...);
    }
    MPI_File_close(...);
}
```

A trace of executing this program results in a compressed trace file consisting of RSD1:$\{100, MPI\_File\_write\}$ and PRSD1: $\{10, MPI\_File\_open, RSD1, MPI\_File\_close\}$. RSD1 captures the inner-loop of MPI_File_write events over 100 iterations. PRSD1 denotes the outer loop for 10 iterations with MPI_File_open/close events and the inner loop as RSD1. Calling contexts of events (signatures of stack back-traces) allow the distinction between different calling sequences in applications. Inter-node compression is performed over a radix tree to unify event parameters for calls. The output trace is a single file of nearly constant size with sufficient information to represent all tasks. Parameters of I/O events are captured as elastic data element representations in ScalaTrace V2 [24], which represents trace data as a list of $< valuevector, ranklist >$ pairs subject to compression.
ScalaTrace records delta times of computation durations between adjacent trace events instead of recording absolute timestamps [18], [15], [25], [19]. Optionally, delta time capturing the duration of an event is recorded as well. Delta time is concisely represented as statistical data of maximum, minimum, average, and variance of delta times and, to provide more detail, also as histograms. During event replay, randomly picked histogram times are emulated to offset native execution of MPI events with their parameters. The timing of replays thus closely resembles that of the original application.
ScalaExtrap[23] exploits a set of algorithms and techniques to extrapolate full communication traces and execution times of an application at larger scale. Since topology is the basis of communication trace extrapolation, ScalaExtrap focuses on identifying the communication pattern of mesh/stencil patterns by calculating the dimension and corner node of the communication stencil. To extrapolate a communication parameter, ScalaExtrap constructs a number of linear equations to indicate how the topology information is related to the parameter by employing Gaussian Elimination to solve the equations.
ScalaTrace preserves the delta time between two events and records the time as multi-bin histograms and extrapolates the timing information of the application via curve fitting using four statistical models for each extrapolation: (1) constant, (2) linearly increasing/decreasing, (3) inverse proportional, and (4) inverse proportional plus some constant.
ScalaTrace preserves the delta time between two events and records the time as multi-bin histograms. Such histograms contain the overall average, minimum, and maximum delta time. ScalaExtrap also extrapolates the timing information of the application via curve fitting to capture variation trends of delta times with respect to the number of nodes as $t = f(n)$, where t is the delta execution time and $n$ is the total number of nodes. ScalaExtrap utilizes four statistical models based on curve fitting for each extrapolation, defines a accuracy metric, $d_i$, per model, and then selects model $i$ best on the best fit, $min(d_i)$:
1) Constant: This method captures constant time $t = f(n) = c$, and $d_1 = std.dev./average$ is used for identification.
2) Linear: This method captures linearly increasing/decreasing trends $t = f(n) = an + b$. For curve classification, 1) $d_2 = \sqrt{residual}/average$, where average refers to the average value of the estimated running times, and 2) a threshold slope $s_m = 0.2$ such that $\forall a < s_m\ t = f(n) = b$, are used.
3) Inverse Proportional: This method captures inverse-proportional trends $t = f(n) = k/n$. To capture the fitting curve, ScalaExtrap calculates the standard deviation of $k_i = t_i \times n_i$ and then divides by the average value of $k_i$, where $d_3 = std.dev./average$ is used for identification.
4) Inverse Proportional+Constant: This method captures the time consisting of an inverse proportional phase and a constant phase $t = f(n) = k/n+c$ and uses $d_4 = \sqrt{residual}/average$ for identification.

The Co-design of Exascale Storage System (CODES) framework [10], [11] is designed for evaluating exascale storage system design points. The CODES framework explores the Rensselaer Optimistic Simulation System (ROSS), which is a discrete-event simulation framework allowing simulations to be run in parallel [28], [26], [2]. CODES and ROSS have been used to construct models of the PVFS file system and the I/O subsystem of the Intrepid IBM BG/P platform in previous work. CODES can model 1) network behavior, 2) hardware components and 3) software protocols. The CODES storage system simulator supports the necessary protocol to handle application-level file open, close, read, and write operations.

## III. Design And Implementation

I/O analysis is a challenge due to multi-layer I/O stacks and multiple I/O patterns of programs. In this section, we introduce (a) capabilities for trace compression, (b) analysis of the trace and extrapolation into target sizes of nodes, and (c) replay capabilities on elastic data representations of MPI-IO and POSIX I/O function calls. In contrast to MPI

communication tracing and past work on extrapolation, we propose a number of novel tracing techniques necessitated by the unique characteristics of parallel I/O.

We design the *ScalaIOTrace*, *ScalaIOExtrap* and *ScalaIOReplay* tools suitable for single program multiple data (SPMD) programs with mesh/stencil communication patterns. Each I/O call is regarded as an event, and sequences of such events are represented as a PRSD using the techniques of ScalaTrace, ScalaExtrap, and ScalaReplay. Hence, this work focuses on the parameter level of I/O events.

### A. ScalaIOTrace

Lossless tracing is imperative for accurate replay. We record the delta time between events and I/O calls with all parameters, except for the actual data that is read/written to a file system. Applications may interleave MPI-IO (for parallel I/O) with I/O syscalls, depending on the software layer. Our objective is to trace and compress I/O at all levels and preserve event ordering. Yet, different interpositioning techniques are required per level. **MPI-IO** is intercepted at the MPI profiling layer (PMPI). PMPI wrappers trace all parameters of MPI-IO calls, but some require domain-specific compression detailed later. **POSIX I/O** at a lower level is captured via GNU link time entry interpositioning with domain-specific parameter compression (using a "__wrap_ " syntax) resembling that of PMPI. Inside wrappers, parameters are collected and compressed before the actual POSIX I/O call ("__real_ ") is invoked. Notice that MPI-IO often uses POSIX I/O to implement its primitives. Wrapping both layers allows us to detect if a lower layer (POSIX I/O) call is made within one of the upper layer (MPI-IO) so that inner calls are not replayed (even though they are traced) as outer ones provide a richer semantics.

### B. ScalaIOExtrap

In order to meet the objective of rapidly obtaining the I/O behavior of parallel applications at arbitrary scale without actual execution, we developed $ScalaIOExtrap$. We exploit different methods for different types of parameters based on their characteristics, e.g., for string-based parameters such as filenames and data-based parameters such as offsets. ScalaTrace is a lossless and scalable tracing tool. The challenge of $ScalaIOExtrap$ is how to maintain the properties of ScalaTrace. We need to extrapolate all processors with exact parameters. ScalaTrace will generate an identical pattern in a trace for most SPMD programs regardless of the number of ranks. For extrapolation, we utilize four trace files of smaller size as input and assure that they have the same number of events.

*1) High-level extrapolation:* Since we assume the patterns of trace files generated from a SPMD program to be identical irrespective of the number processors it runs on, we maintain the event numbers and event names. For example, if the $n_0$th event is MPI_File_open for input trace files, then we also generate an MPI_File_open as the $n_0$th event for the target trace file. ScalaTrace records rank lists at the event level. We exploit Gaussian Elimination introduced in related work to extrapolate these ranklists for mesh/stencil communication patterns [23].

Another aspect to be considered, which is unique to ScalaTrace, is loop iteration. For scalablility, ScalaTrace uses RSDs during intra-node-compression to generate a loop number recording the iteration times of each event. For *weak scaling* (where the workload assigned to each processor stays constant as number of processors increases) extrapolation is easy, e.g., each rank reads N bytes no matter how many ranks are running, and the loop iterations will not change regardless of rank size. However, under *strong scaling* (where the total workload is fixed, i.e., the workload assigned to each processor decreases as number of processors increases) and also for tracing the lower level POSIX-IO for collective MPI-IO calls [4], loop iterations will change. In most cases, loop iterations will be inverse proportional to the size of ranks. We construct a set of equations based on the number of ranks and loop iterations to determine their exact relationship and calculate the loop iterations for a target number of ranks.

*2) Elastic string extrapolation:* Extrapolation for strings, especially filenames, is important in ScalaIOExtrap. Filenames play a major role in distinguishing different I/O patterns (see Section I). For pattern A (Serial I/O) and C (Parallel I/O, shared-file), extrapolated filenames are identical regardless of the number of ranks. Hence, we also generate the same filenames as for trace files of smaller number of ranks.

For pattern B (Parallel I/O with one file per process, N-to-N), filenames are traced and compressed as an RSD $[start\ stride\ size]$ pattern. We assume the variables in filenames have a linear relationship to the rank numbers, which is common for the N-to-N pattern and even the N-to-N/n pattern. Example: **a) N-to-N pattern:** If the filename in the program is "`/dir0/file_<rank>`", the variable is `<rank>` and it has a linear relationship to rank numbers, $variable = 1 \times rank + 0$. **b) N-to-N/n pattern:** This means all the ranks are gathered as groups, and each root of the group acts the "spokesperson" performing I/O. The variables also have a linear relationship to rank numbers. Example: A program with four ranks acting as a group has a filename "`/dir0/file_<rank/4>`", i.e., the variable also has a linear relationship to the ranks. With this assumption, we determine that $start$, $stride$, $size$ of an RSD pattern have a linear relationship to rank size. In most cases, the $start\ and\ stride$ do not change (as we observe in experiments), only $size$ changes with rank size. We simply generate the equations over the reference of traces and solve them using Gaussian Elimination.

TABLE I: Offset parameters for Rank0-Rank5

| Rank | size | i=0 | i=1 | i=2 | size | i=0 | i=1 | i=2 |
|---|---|---|---|---|---|---|---|---|
| Rank0 | | 0 | 960 | 1920 | | 0 | 960 | 1920 |
| Rank1 | | 240 | 1200 | 2160 | | 160 | 1120 | 2080 |
| Rank2 | 4 | 480 | 1440 | 2400 | 6 | 320 | 1280 | 2240 |
| Rank3 | | 720 | 1680 | 2640 | | 480 | 1440 | 2400 |
| Rank4 | | - | - | - | | 640 | 1600 | 2560 |
| Rank5 | | - | - | - | | 800 | 1769 | 2720 |

*3) Elastic data element extrapolation:* Elastic data, such as $offset$ and $count$, are the most challenging to extrapolate since (a) we do not know a mathematical model and (b) the two dimensions of matrix data need to be extrapolated, and (c) we want to extrapolate exact data for all ranks at target size.

We first motivate the two dimensions of the matrix data. Since ScalaTrace can perfectly compress trace data both intra-node and inter-node, the following strong scaling code will generate the offset parameter in Table I after compression.

```
for (int i=0; i<3; i++){
  offset = rank*(960/rank_size)+960*i;
  MPI_File_seek(...offset...);
}
```

For the column dimension, the values of each column in Table I denote offsets for different ranks for the same loop iteration while values per row are offsets of the same rank number for different loop iterations. Consider an attempt to extrapolate to 8 ranks: (a) If we only extrapolated in column dimension, we would not know the value for different loop iterations. (b) If we only extrapolated in row dimension, we would not have data for Rank 6 and Rank 7. Hence, extrapolation needs to span all dimensions of the iteration at once. We create a mathematical model for column extrapolation using the four models introduced in Section II plus a new model:

5) $offset = ((rank + a)\%RankSize) \times b$, where a (rank offset) and b (rank stride) are constants, rank is the rank number and RankSize is the number of ranks.

We require that the standard deviation of a singular model (1-5) is zero as we need an exact solution, not an approximate one, to extrapolate trace data. (This is a deviation from the models in Section II, which used the smallest standard deviation as identification for trace generation, but now these models are reused and augmented with model 5) for extrapolation.) If we cannot find a model with zero standard deviation, we flag the base traces to invalid for extrapolation by our method. While our models cover the common cases, an interface is provided for users to add their own models for extrapolation by specifying their own model (function) as a plug-in. Notice that in later the experiments, no plug-ins were required.

We extrapolate the column dimension for both weak scaling and strong scaling: Weak scaling is simple since the values will be same for the different rank sizes. For strong scaling, we also use the $k/n + c$ model to predict the results for a target rank size. After obtaining the first parameters from column extrapolation, we combine them with the row extrapolation equation and then calculate the remaining parameters.

*4) Handles and time extrapolation:* As mentioned in Section III-A, handles are coded into integers. Here, we use the same technique as for extrapolating data-based parameters. Normally, handle extrapolation is simple. E.g., for a file handle, in either the SIO, N-to-N, or N-to-1 I/O pattern, all ranks perform the same file open operation regardless of rank size, which remains unchanged during extrapolation. For strong scaling, the open operation depends on rank size. We address this with model-based extrapolation as before. We further reuse time extrapolation by mathematical modeling (see [23]).

### C. ScalaIOReplay

ScalaIOReplay provides time-accurate as well as fast-forward replay options unique to I/O requirements. A parallel trace replay of all events across task ranks preserves per-rank ordering of events. Hence, its replays preserve the I/O semantics of the original application and may also serve as a means to verify the correctness of the tracing framework.

*1) Bridge from ScalaIOTrace to CODES:* Replaying the extrapolated trace file in a simulation environment allows us to experiment with replaying traces on future extreme-scale systems. We integrate our approach into the CODES exascale storage system simulation toolkit to enable extrapolated traces to be used in a generic replay environment. To enable CODES to read the ScalaIOTrace file format, we built a bridge (shown in Fig. 2) between ScalaIOReplay and CODES to convert the ScalaTrace data structure to the CODES data structure. CODES is a POSIX-IO-based simulator. To preserve consistency, we ignore the higher level in the I/O stack [21] and trace lower-level POSIX-IO only during MPI-IO calls with ScalaIOTrace (Multi-level tracing would otherwise result in nested events, which are not supported). Let us describe the concepts of using the bridge: (1) The unique file id for the CODES workload API is an integer, but for ScalaIOReplay, the filename is a unique string (see ScalaIOReplay). We exploit the Jenkins lookup3 hash to generate a hash integer from the string. (2) ScalaIOTrace does not record the offset of files if there is no offset parameter in the calls, but the CODES workload API requires the offset for each I/O call. We record the offset during open in ScalaIOTrace, and update the offset for each file operation to ensure that CODES gets the correct offsets. (3) ScalaTrace is the backbone of ScalaIOTrace. So ScalaIOTrace traces both I/O events and MPI communication events, while CODES currently only needs I/O events. MPI communication events are processed as follows: generate a synchronization event for each MPI_Barrier, and generate a corresponding delay for other communication events.

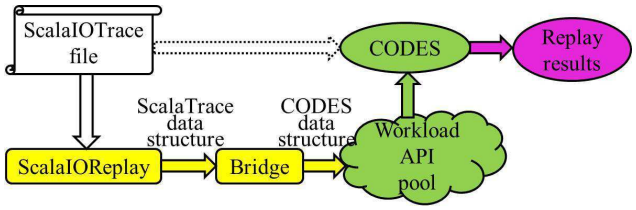


Fig. 2: Bridge between ScalaIOReplay and CODES

The bridge is not only used for connecting ScalaIOTrace traces with CODES, but also for ScalaIOExtrap traces. Since any trace for CODES needs to contain only lower-level POSIX-IO information regardless of whether the application uses POSIX-IO or MPI-IO, processing of events at the bridge level is trivial for applications with only POSIX-IO and N-to-N (MPI-IO) patterns, because the lower-level POSIX-IO behavior is identical to the upper level (e.g., each POSIX read or MPI N-to-N read generates a POSIX read). We focus on how to process MPI collective I/O. Using collective buffering optimizations, MPI-IO gathers all I/O requests and lets only the aggregator issue I/O at the POSIX level, which we called "spokesperson"/proxy in Section I. However, the aggregator may issue large volumes, which presents a challenge. Consider a collective MPI-IO call MPI_File_write() with count=4096, which means each processor is writing 4096 bytes to a single file. After applying the collective buffering optimization, the POSIX-IO information is aggregated as a write of size 4096*16, 4096*32, 4096*48, 4096*64 bytes for RS=16, 32, 48, and 64. Intuitively, we should extrapolate to a single write with count=4096*1024 for RS=1024. However, this is not the case in practice. Instead, 16 different aggregators write the 4096*1024 bytes in parallel due to internal buffer thresholds. To solve this problem, we define a max-blocksize for the bridge. This max-blocksize depends on the configuration of MPI, e.g., 262,144 bytes for MPI by default on the BG/Q platform. Whenever the I/O volume exceeds max-blocksize, we separate a single I/O call into multiple I/O operations.

## IV. Experimental Framework

As mentioned in Section I, to verify the correctness of our approach, we deployed our framework on top of: (1) A smaller cluster (x86_64) with 1728 cores on 108 compute nodes. It supports the Network File System (NFS), Parallel Virtual File System (PVFS2), and a local filesystem. (2) An IBM Blue Gene/Q system which uses the IBM General Parallel File System (GPFS). (3) CODES (Simulator): Co-design of Exascale Storage system [10], [11]. It supports the Parallel Virtual File System (PVFS/BGP) [10]. We evaluate the correctness of our approach for the three platforms using the techniques shown in Fig. 4, Fig. 3 and Fig. 5, respectively, with respect to the following aspects:

$C1$ : We compare the extrapolated trace file with the trace file gathered from ScalaTrace, which is executed at the extrapolated target rank size. Ideally, the two traces file should be exactly the same (after filtering out minor differences in delta time). However, the delta execution time we extrapolated will have some variance. So we ignore the execution time and then compare the structure of the two trace files.

$C2$ : We replay all I/O events (i.e., issue I/O calls natively) using the extrapolated trace file and compare the execution time with the original program running on same number of processors. The two (delta) execution times should be a close match. We simulate communication and computation events during replay via $usleep$. Discussion: We could alternatively replay communication as well. If we did, then overall accuracy would be better than just reporting I/O, i.e., averaging communication+I/O actually skews results to increase accuracy. But we believe that pure I/O extrapolation gives a more honest indication on the limits of our approach.

$C3$ : We use **Darshan** tracing [4], [3] to gather the total I/O size of the original program and the corresponding replayed program. We compare the I/O size difference by eliminating the I/O overhead of the replay engine.

$C4$ : We feed both the extrapolated traces and traces captured from ScalaIOTrace with an identical number of ranks into the CODES simulator and compare the total number of opens, reads, and writes in experiments.

We chose the following I/O benchmarks and mini-applications with I/O:

**IO-sample** (Argonne National Laboratory) features a number of benchmarks including POSIX-IO (an N-to-N pattern), MPI-IO (shared N-to-1), and MPI-IO (N-to-N) with calls of derived I/O datatypes and a variety of I/O calls.

**Interleaved Or Random (IOR)** (Lawrence Livermore National Laboratory) is used for performance testing of parallel file systems for high performance clusters. IOR provides the interface for users to verify the overall I/O size, individual transfer size, file access mode (single shared file, one file per processor), and whether the data is accessed using a chunk pattern or an interleaved pattern.

Based on the characteristics of the platforms, we verify our results differently in different cluster as explained next.

### A. Commodity Linux Cluster

We first conduct experiments on a commodity Linux cluster of 108 nodes, where a node has two AMD Opteron 6128 processors with 8 cores each (16 per node) and an InfiniBand interconnect between nodes. We varied the number of target processors during I/O extrapolation and replayed with a corresponding number of nodes. An identical configuration is important since I/O bandwidth and contention depend on the number of tasks per node and the total number of nodes. The filesystem type also impacts I/O behavior, as our experiments cover a local filesystem, a shared network filesystem (NFS), and a Parallel Virtual File System (PVFS2). Fig. 3 depicts the set of verification experiments conducted on local, NFS, and PVFS2 filesystems with the same I/O application and the same input parameters (e.g., I/O Size, I/O pattern) using methods C1 and C2 (as the others are not applicable).

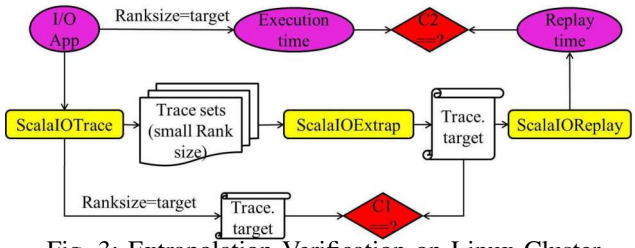


Fig. 3: Extrapolation Verification on Linux Cluster

### B. IBM BG/Q

We also conduct experiments on an IBM BG/Q system with the configuration shown in TABLE II. We also used the

TABLE II: Configuration of IBM BG/Q system

| | |
|---|---|
| Architecture: IBM BG/Q | Cabinets: 4 |
| Processor: 16 1600 MHz PowerPC A2 cores | Nodes: 4096 |
| Total cores: 65,536 cores | Cores/node: 16 |
| Memory/node: 16 GB RAM per node | Memory/core: 1 GB |

Darshan I/O characterization tool to record statistics such as the number of files opened, time spent in performing I/O, and the amount of data accessed by an application. We further analyze Darshan logs by using the "darshan job summary" utility. Finally, we verify the correctness of experiments on the BG/Q platform as shown in Fig. 4 using methods C1/C2/C3 (as C4 is not applicable). Trace files are 2KB to 10s of KB in size (due to very effective structural compression via PRSDs).

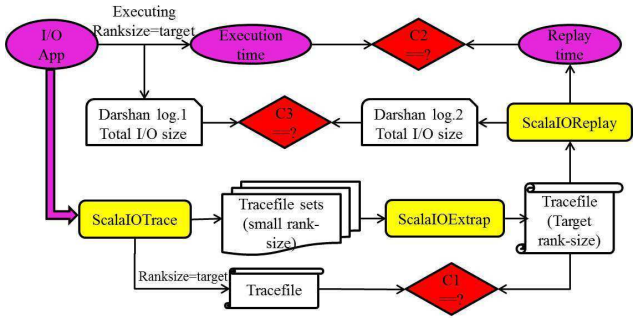


Fig. 4: Extrapolation Verification on BG/Q

### C. CODES

ROSS [27] offers two methods of simulation: the optimistic and the conservative mode. The optimistic mode is a fast mode, where "'reverse function handlers" execute events out of order without violating any timing dependency. However, the optimistic mode is not supported for BG/P. Hence, we utilize the conservative mode without reverse handlers. More specifically, we conduct simulations on a x86_64 machine, execute events in order, and capture statistics. By using the bridge from ScalaIOTrace to CODES, we can feed ScalaTrace traces into the CODES simulator to support the verification method shown in Fig. 5.

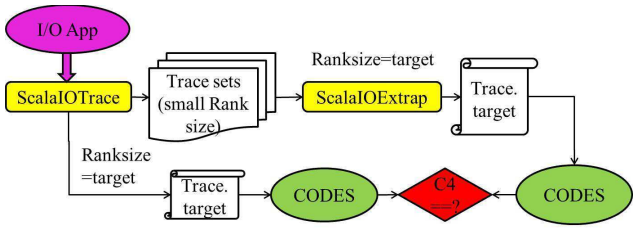

Fig. 5: Extrapolation Verification on CODES

## V. Results

We compare the traces, total I/O volume, statistics and execution time (e.g., number of total open, read, write, close operations) of our purposed approach for 16 processes per node. It is straightforward to compare the first three results, since no matter how the environment changes, they are fixed for identical input parameters and number of ranks. However, execution time comparison is complicated due to significant time variations even in the same environment and with the same I/O application due to contention. A residual difference between the extrapolated trace replay time and observed execution time is deemed to exist. This difference is calculated as $abs(T_{extrapolated} - T_{observed})/T_{observed}$, where $T_{extrapolated}$ is the replay time of the extrapolated traces and $T_{observed}$ is the execution time of the I/O application with the same number of ranks. In order to minimize contention, we only run one experiment at a time and collect execution time by averaging three captured runs.

### A. Results on the Linux Cluster

To verify the correctness as well as the accuracy of ScalaIOExtrap, we conducted our experiments on various filesystems. As shown in Fig. 3, we compare the traces and execution times with the filesystems: (1) The local Filesystem using local data storage devices, (2) the Network Filesystem (NFS), a shared filesystem that allows a user on a client computer to access files over a network, and (3) the Parallel Virtual File System (PVFS2), a file system that distributes its data over multiple storage nodes.

**IO-sample:** The IO-sample benchmark features both MPI-IO and POSIX-IO, as well as the N-to-1 and N-to-N patterns. Our replay engine can reconstruct the original benchmark irrespective of I/O patterns and supported I/O libraries. For a set of input parameters with POSIX-IO (I/O size: 8KB per processor; iterations: 100) and MPI-IO (iterations: 3; I/O size : 1M, 2M and 3M bytes per rank, i.e., 8G/16G/24G total), we gathered traces for 8, 16, 24, and 32 ranks, which comprises the set of small traces. From the small traces, we extrapolate and generate traces for 128, 192, 256, and 320 ranks. We use the UNIX diff utility to compare the ScalaIOTrace and extrapolated traces with the same number of ranks. Except for the time extrapolations, they match perfectly. Timings (y-axis) are shown for different number of ranks (x-axis) in Fig. 6. For the same I/O size and I/O pattern, the local filesystem takes the shortest time because it is faster for nodes to access their local memory. PVFS2 takes the longest time (explained later for IOR results). The replay time fluctuates with application execution time. Time inaccuracy is within 5%.

**IOR:** IOR is a more complex I/O benchmark. We capture results for different inputs classified as shared-file (chunk pattern), shared-file (interleaved pattern), and file-per-processor.

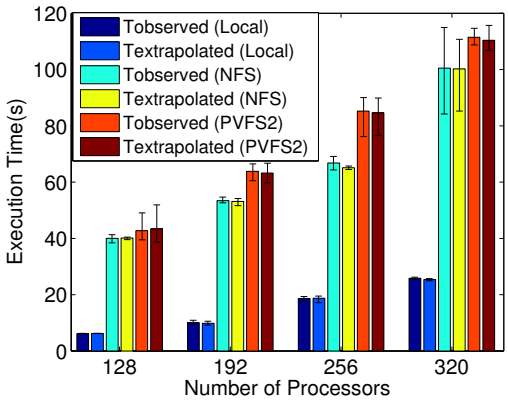

Fig. 6: Results of IO-sample in Local, NFS and PVFS2

Both shared-file cases (chunk and interleaved) follow a N-to-1 pattern. They differ in how they order I/O. Consider four processors, A, B, C, and D, each of them performing four I/O operations. Shared-file (chunk pattern) performs I/O as AAAABBBBCCCCDDDD, while shared-file (interleaved pattern) performs I/O as ABCDABCDABCDABCD. We select two patterns since they differ in whether they use the collective buffering or not. MPI-IO allows users to access non-contiguous data with a single I/O function call [20]. The interleaved pattern contains many small, distinct I/O requests that are densely interleaved, so that MPI-IO uses collective buffering to transfer the data to the file system from an aggregator for larger I/O chunks. E.g., if A is the aggregator, then B, C, and D will send their data to A, and A will write it as one big chunk. The chunk pattern, in contrast, has a big gap between written file regions, so MPI-IO has no choice but to issue them as individual operations to the file system.

We select the I/O size to be TransferSize=128K, and each processor accesses 2M data. As in the IO-sample benchmark, we use the UNIX diff utility to compare the traces gathered from ScalaIOExtrap and ScalaIOTrace for the same number of ranks. They matched perfectly. We depict the execution times of IOR (shared-file) in Fig. 7 and Fig. 8.

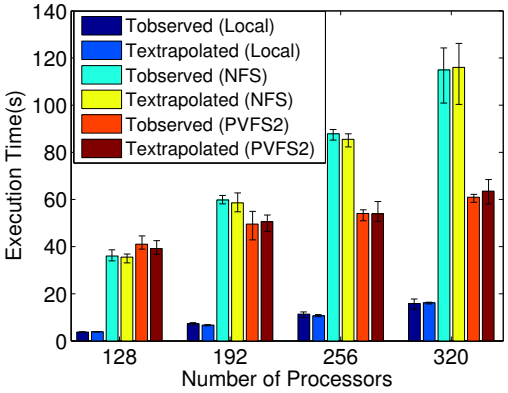

Fig. 7: Results of IOR (chunk pattern)

Similar to IO-sample, the execution times for the local file system are the shortest among all file systems. Chunk/interleaved patterns do not differ for local storage, because nodes access private resources. The crux is to capture the change in patterns when scaling, not the size, as I/O is replayed (takes the same time). Execution time increases due to computation replayed as delta-time delays, but because of I/O.

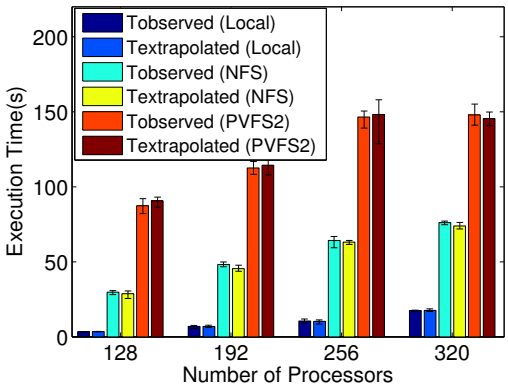


Fig. 8: Results of IOR (interleaved pattern)

We also obtain timing results per processor for the N-to-N pattern (see Fig. 9). When comparing the results shown in Fig. 7, Fig. 8 and Fig. 9, PVFS2 performs best in terms of per processor time because N-to-N I/O has the highest degree of parallelism. Although I/O patterns and file systems are varied and even though different extrapolation techniques are needed for different I/O patterns, our extrapolated results match the actual application. By avoiding contention as much as possible, we obtain time accuracy within $5\%$, which means our approach reflects the behavior of applications quite well.

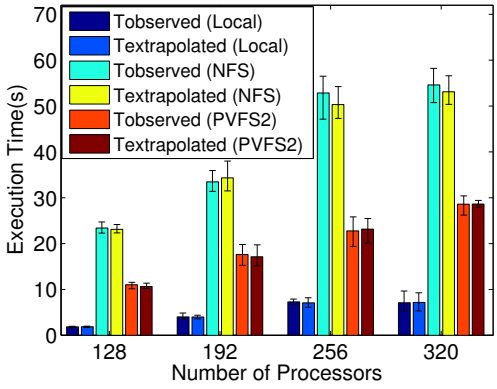


Fig. 9: Results of IOR file-per-processor

### B. Results on BG/Q

We conduct experiments with 16 processes per node (one per core). Before comparing traces and execution time, we assess the total I/O size (see Fig. 4) using the Darshan tracing tool. By parsing the Darshan log, we obtain the total I/O size of all reads and writes over all processors. ScalaIOReplay issues its own I/O to read in traces to memory, which introduces overhead proportional to its size. We capture the extrapolated I/O size as: $S_{replayed} - (S_{trace} \times RS)$, where $S_{replayed}$ is the replayed I/O size captured by Darshan, $S_{trace}$ is the size of trace read by ScalaIORelay, and $RS$ is the number of ranks. To emphasize differences for large numbers of ranks, we separately report IO-sample results for POSIX-IO (N-to-N pattern) and MPI-IO (N-to-1 pattern). After selecting input parameters for POSIX-IO (I/O size: 4KB per processor; iteration times: 100) and MPI-IO (iteration times: 2; I/O size: 64KB and 128KB), we gather traces from 16, 32, 48, and 64 ranks as the small trace set for extrapolation. Execution time results depicted in Fig. 10 (logarithmic y-axis) indicate an average time difference of no more than $5\%$. We also gather IOR execution time results for shared-file (interleaved),

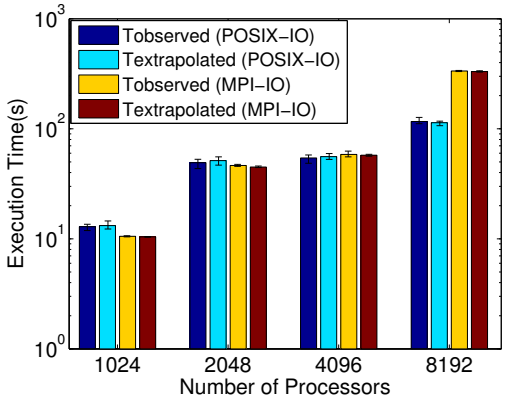


Fig. 10: Results of IO-sample on BG/Q

shared-file (chunk), and file-per-processor patterns. We set the TransferSize=8K and let each processor issue 64KB of I/O. The execution time results of IOR are shown in Fig. 11. The BG/Q system uses the General Parallel File System (GPFS), which is a high-performance shared-disk clustered file system. GPFS performs better for large I/O operations as seen in Fig. 11 (again with a logarithmic y-axis). For the same I/O size and with MPI-IO collective buffering (see Section V-A), the shared-file (interleaved) pattern is significantly faster than the shared-file (chunk) pattern, while file-per-processor takes the longest time. For both the IO-sample and IOR benchmarks, we also use the UNIX diff utility to compare corresponding traces (same number of ranks and same input parameters) captured from ScalaIOTrace and ScalaIOExtrap. Except for the recorded time, they match perfectly. The execution time results indicate that the replay time of the extrapolated trace accurately reflects the execution time of the original application.

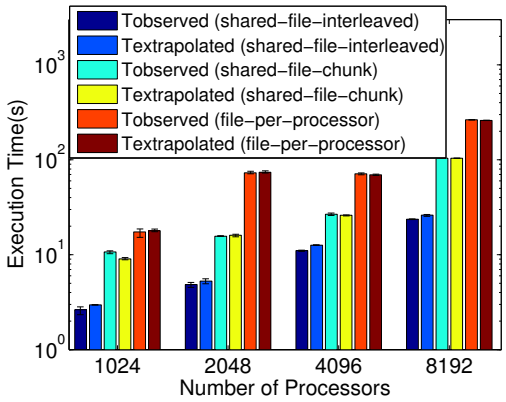


Fig. 11: Results of IOR on BG/Q

We assess how closely our ScalaIOExtrap matches the original I/O size using the verification from Fig. 4. After eliminating the ScalaIOReplay overhead, results in Fig. 12 indicate that the total I/O size gathered from ScalaIOReplay is exactly same as that gathered from an actual application run.

### C. Results for CODES

CODES, the virtual HPC simulation system, provides statistics of the total number of I/O calls per operation (open, read, write, close, and synchronization). We already compared the extrapolated results at the structural level. We now compare the results at the operational level as shown in Fig. 5. Our goal is to verify the correctness of our approach. Although we can extrapolate traces to extremely large numbers of ranks and

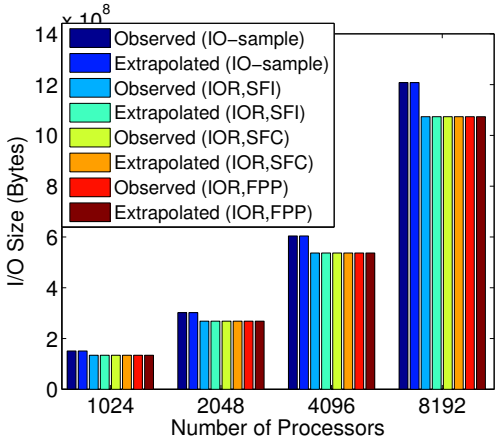


Fig. 12: I/O size of IO-sample, IOR shared-file-interleaved (SFI), IOR shared-file-chunk (SFC) and IOR file-per-processor (FPP) on BG/Q

simulate it through CODES, we extrapolate to more moderate numbers of ranks to be able to compare to traces gathered on actual HPC systems. We distinguish the I/O patterns POSIX-IO (N-to-N), MPI collective I/O (N-to-1, SIO, N-to-N/n), and MPI file-per-processor (N-to-N). We capture POSIX-IO results for IO-sample (see Fig. 13 with a logarithmic y-axis). We compare the total number of opens, reads, writes, and closes over all processors between the original and extrapolated traces. Let $N$ be the number of operations in the $Observed$ (application) and $Extrapolated$ (replayed) traces, e.g., $Observed\ Nreads$ is the total number of reads of the application. We observe that observed and extrapolated results are identical for POSIX-IO.

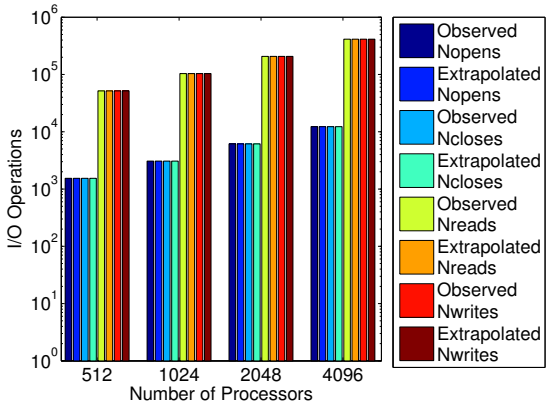


Fig. 13: I/O operations of IO-sample (POSIX-IO)

Collective MPI-IO contains N-to-1, SIO, and N-to-N/n patterns. (1) For MPI-IO, the N-to-1 pattern reduces processors to I/O in a single file. (2) For POSIX-IO, if the total I/O workload is small enough for a processor, one processor acts as the aggregator, which becomes Serial I/O (SIO). (3) For POSIX-IO, if the total I/O workload is large, N/n aggregators are needed, which is the N-to-N/n pattern. By setting the max-blocksize to 256KB, we obtain the collective MPI-IO results depicted in Fig. 14 (logarithmic y-axis). The extrapolated number of I/O operations perfectly matches the number of observed I/O operations.

We compare the results with and without max-blocksize to further illustrate the impact of this parameter. Fig. 15 depicts the difference for extrapolations to 512, 1024, 2048, and 4096 ranks. The max-blocksize does not effect the number of open and close calls, so we only compare the number of reads and

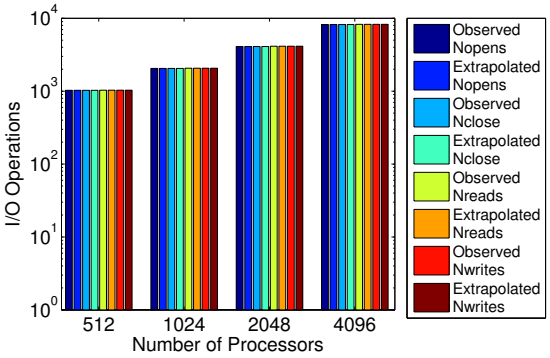


Fig. 14: I/O operations of IO-sample (MPI collective I/O)

writes, where $R$ is the observed number of reads; $Rw$ is the extrapolated number of reads with max-blocksize; $Rw/o$ is the extrapolated number of reads without max-blocksize; $W$ is the observed number of writes; $Ww$ is the extrapolated number of writes with max-blocksize; and $Ww/o$ is the extrapolated number of writes without max-blocksize. Results indicate that

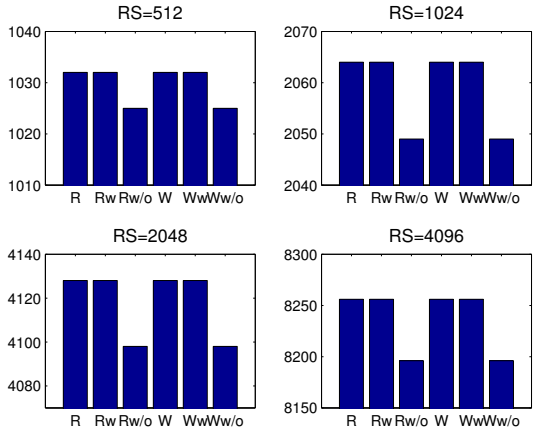


Fig. 15: I/O operations of with and without max-blocksize (IO-sample, MPI collective I/O)

the difference increases as the number of ranks, RS, increases. The max-blocksize parameter does not effect the total I/O workload in size, but it adversely affects the performance of the N-to-1 and N-to-N/n patterns.

We next capture IOR results in file-per-processor mode (see Fig. 16, logarithmic y-axis). Results indicate that under the

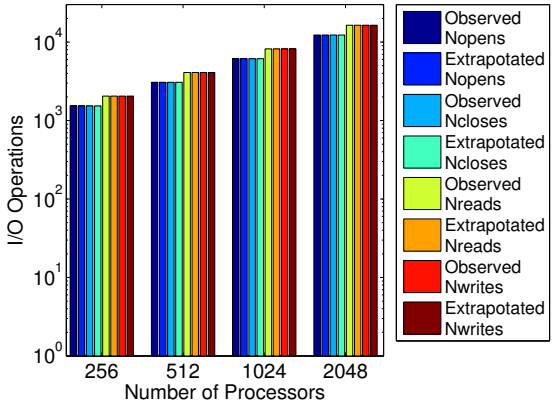


Fig. 16: I/O operations of IOR (file-per-processor)

file-per-processor (N-to-N) mode, our approach extrapolates traces correctly as well.

### D. Results for ddcMD

The ddcMD multi-physics particle dynamics code is an scalable code for modeling particle dynamics and includes advances in high performance scalable I/O. We use a standard version of ddcMD and its regular checkpoint capabilities at timestep intervals in a k-to-1 pattern, where k aggregators write to 1 file. Groups of N/k nodes each send their checkpoint data to their respective aggregator. Each rank's I/O size is 16K before aggregation. In order to focus on I/O tracing and extrapolation, we use $usleep$ to simulate communication and computation events during replay.

There are three types of non-IO events that need to be considered for ddcMD during delta-time extrapolation:

- Communication events: These include all MPI communication functions in ddcMD, namely MPI_Send, MPI_Recv, MPI_Isend, MPI_Irecv, MPI_Wait, and MPI_Waitall.
- Reduce events: MPI_Reduce is used.
- Other events: The time between any other MPI events is calculated as

$$total\_execution\_time - Communication\_time - MPI\_Reduce\_time.$$

This encompasses the time for other MPI events, any computation time, and the differences between (average) recorded time and actual execution time. As the latter may diverge from the average, it is important to compensate for per-rank difference from the recorded average. This is an important finding: extrapolation was found to only be accurate with this compensating term at scale.

For these event types, we record their mean and then extrapolate each of them separately. Since ddcMD requires the number of processors to be powers of two by input constraint, we gather results on ACLF for an increasing sequence of just enough processors to be able to extrapolate. The results for ddcMD, which include C-style IO for checkpointing, are depicted in Fig. 17. ScalaIOExtrap automatically selects

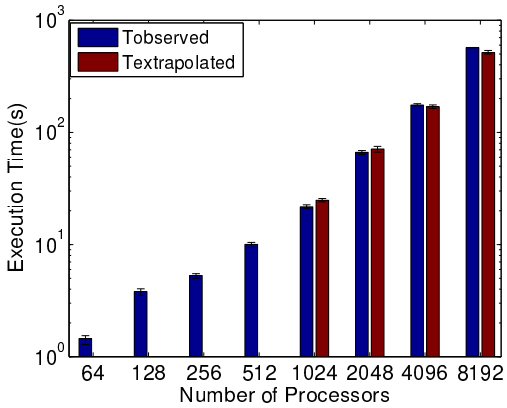


Fig. 17: Results of ddcMD on BG/Q

different curves per event type as a closest fit of a curve for time extrapolation. For example, communication time, $f(x)$, resembles a 2nd-order polynomial, $f(x) = ax^2 + bx + c$, while reduce time follows a 1st-order (linear) relation with respect to the number of processors, $x$. With these models, we observe that as the number of processors increases, the overall extrapolated execution time under replay is shorter than the actual execution time of the application. This is due to the fact that I/O is re-executed during replay while communication events are not replayed, i.e., only their extrapolated time is considered. Hence, any synchronization due to collectives or delay due to blocking point-to-point messages is ignored during replay. Results show that time accuracy is within $15\%$. Trace file are about 30KB in size (constant size from 64-8192 ranks), differences in bytes are due to numbers encoded as ASCII (variable string length), again due to very effective structural compression via PRSDs). The cost of intra- and inter-node compression was 2 and 98 seconds, respectively, for 4k processors. This cost is incurred only once. Thereafter, any number of extrapolations can be generated.

## VI. Related work

Leung et al. [9] proposed an analysis framework based on server-side tracing data. Liu et al. [12] combine filtering techniques with wavelet transforms to characterize noisy server-side I/O behavior in a lossy (approximate) manner. Our method utilizes structural compression of client-side I/O traces in a lossless manner. They trace when and how much data is transmitted; we add files, patterns, and offsets to that, and then extrapolate.

Wright and Hammond [22] analyzed the write bandwidth of MPI-IO as well as POSIX file system calls originating from MPI-IO at increasing scale by utilizing the RIOT toolkit, which is able to capture and record I/O operations of applications. In contrast, we focus on extrapolating I/O traces to arbitrary an number of ranks.

Eckert and Nutt [6], [5] studied the extrapolation of trace data of multiple threaded programs on shared memory multiprocessors. Instead, our work focuses on I/O traces and is based on deterministic application execution, i.e., we preserve the causal orders both for ScalaIOTrace and ScalaIOExtrap.

Mohror and Karavanic [14] assessed different trace reduction techniques. Their similarity metric (performance) resembles our wall-clock time. But their per-core metrics appear to lack scalability. If they were enhanced so that they scaled, they would be similar to our histograms. Their compression reduction (based on flat distances) is inferior to the more general structurally recursive compression of ScalaTrace [24]. But while ScalaTrace/Extrap only traces MPI communication events, ScalaIOTrace/Extrap traces I/O events, which require different extrapolation than communication events (see Section III). Vijayakumar et al. [21] developed fundamental I/O tracing capabilities for ScalaTrace. Luo et al. [13] explored techniques for small-scale extrapolation of I/O traces. Our paper builds on this related work but contributes novel results for larger-scale and multi-platform experiments, including the extrapolation of a single application split into different workload components by disjoint interpolation functions for each component, which was shown to be required for larger applications (ddcMD).

## VII. Conclusion

We presented the design and implementation of three tools:
**(1) ScalaIOTrace** is a multi-level I/O tracing approach, which supports both MPI-IO and POSIX-IO interpositioning. It captures I/O events as singletons, vectors, and RSDs in an elastic trace representation, which is stored in a single, lossless, and order-preserving trace file.
**(2) ScalaIOExtrap** is an extrapolation tool. By analyzing a set of smaller traces, modeling the relation between parameters

and the number of ranks, it calculates parameters and generates a single trace for any number of ranks. Experimental results demonstrate that structural trace comparison, I/O size, and the number of operations remain perfectly accuracy while execution time remains sufficiently accurate.
**(3) ScalaIOReplay** is a parsing and replay engine, which re-executes the events of ScalaIOTrace traces in the same order as an original application. With ScalaIOReplay, users can analyze the application without the need to scale their source code.

Our results demonstrate that our approach is not only scalable in terms of ranks but also works across different platforms. We can generate and then replay extrapolated trace for large number of ranks by collecting a set of traces with smaller number of ranks. We preserve event ordering and time accuracy in these large traces. With this technique, large-scale communication and I/O performance estimations can be obtained without modifying the source code of such applications for these larger scales. We can conclude that our approach opens up new opportunities for I/O performance analysis with good scalability and portability. And from the given experimental results, we can conclude that I/O behavior of parallel applications can be analyzed from a set of smaller traces and extrapolated to a trace of arbitrary number of ranks while retaining correct communication patterns, I/O size, I/O operations, and execution times for mesh-based communication patterns. Future work aims at targeting diverging communication patterns between nodes and fitting other models to scaling trends, e.g., via plug-ins [1].

## ACKNOWLEDGEMENTS

This work was supported in part by NSF grants 0958311, 1058779, 1217748 and by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computer Research (ASCR), under contract DE-AC02-06CH11357. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

## REFERENCES

[1] S. Ananthakrishnan and F. Mueller. Scalajack: Customized scalable tracing with in-situ data analysis. In *Euro-Par Conference*, pages 13–25, Aug. 2014.

[2] 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*, PADS '09, 2009.

[3] P. Carns, K. Harms, R. Latham, and R. Ross. Performance analysis of darshan 2.2. 3 on the cray xe6 platform. Technical report, Argonne National Laboratory (ANL), 2012.

[4] P. H. Carns, R. Latham, R. B. Ross, K. Iskra, S. Lang, and K. Riley. 24/7 characterization of petascale I/O workloads. In *Proceedings of the First Workshop on Interfaces and Abstractions for Scientific Data Storage*, New Orleans, LA, USA, 09/2009 2009.

[5] Z. K. Eckert and G. J. Nutt. Parallel program trace extrapolation. In *Parallel Processing, 1994. Vol. 1. ICPP 1994. International Conference on*, volume 2, pages 103–107. IEEE, 1994.

[6] Z. K. F. Eckert. Trace extrapolation for parallel programs on shared-memory multiprocessors. *Technical Report, Spring 5-1-1996*, 1995.

[7] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, SC '01, 2001.

[8] S. J. Kim, Y. Zhang, S. W. Son, R. Prabhakar, M. Kandemir, C. Patrick, W.-k. Liao, and A. Choudhary. Automated tracing of I/O stack. In *EuroMPI*, 2010.

[9] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. In *USENIX 2008 Annual Technical Conference*, ATC'08, 2008.

[10] N. Liu, C. Carothers, J. Cope, P. Carns, R. Ross, A. Crume, and C. Maltzahn. Modeling a leadership-scale storage system. In *Proceedings of the 9th International Conference on Parallel Processing and Applied Mathematics - Volume Part I*, PPAM'11, 2012.

[11] 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 *In Proceedings of the 2012 IEEE Conference on Massive Data Storage*, 2012.

[12] Y. Liu, R. Gunasekaran, X. Ma, and S. S. Vazhkudai. Automatic identification of application I/O signatures from noisy server-side traces. In *FAST*, pages 213–228, 2014.

[13] X. Luo, F. Mueller, P. Carns, J. Jenkins, R. Latham, R. Ross, and S. Snyder. Hpc i/o trace extrapolation. In *Workshop on Extreme-Scale Programming Tools*, Nov. 2015.

[14] K. Mohror and K. Karavanic. Evaluating similarity-based trace reduction techniques for scalable performance analysis. In *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on*, pages 1–12, Nov 2009.

[15] F. Mueller, X. Wu, M. Schulz, B. R. De Supinski, and T. Gamblin. Scalatrace: tracing, analysis and modeling of HPC codes at scale. In *Applied Parallel and Scientific Computing*, pages 410–418. Springer, 2012.

[16] M. Noeth, F. Mueller, M. Schulz, and B. R. De Supinski. Scalable compression and replay of communication traces in massively parallel environments. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–11. IEEE, 2007.

[17] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. de Supinski. Scalatrace: Scalable compression and replay of communication traces for high-performance computing. *Journal of Parallel and Distributed Computing*, 69(8):696–710, 2009.

[18] P. Ratn, F. Mueller, B. R. de Supinski, and M. Schulz. Preserving time in large-scale communication traces. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 46–55. ACM, 2008.

[19] D. A. Reed, P. Roth, R. A. Aydt, K. Shields, L. Tavera, R. Noe, and B. Schwartz. Scalable performance analysis: The Pablo performance analysis environment. In *Scalable Parallel Libraries Conference, 1993., Proceedings of the*, pages 104–113. IEEE, 1993.

[20] R. Thakur, W. Gropp, and E. Lusk. Optimizing noncontiguous accesses in mpi – io. *Parallel Comput.*, 28(1):83–105, Jan. 2002.

[21] 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*, PDSW '09, 2009.

[22] S. A. Wright, S. D. Hammond, S. J. Pennycook, and S. A. Jarvis. Light-weight parallel I/O analysis at scale. In *Computer Performance Engineering*, pages 235–249. Springer, 2011.

[23] X. Wu and F. Mueller. Scalaextrap: Trace-based communication extrapolation for spmd programs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 113–122, Feb. 2011.

[24] X. Wu and F. Mueller. Elastic and scalable tracing and accurate replay of non-deterministic events. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, 2013.

[25] X. Wu, K. Vijayakumar, F. Mueller, X. Ma, and P. C. Roth. Probabilistic communication and I/O tracing with deterministic replay at scale. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, 2011.

[26] G. Yaun, C. D. Carothers, and S. Kalyanaraman. Large-scale tcp models using optimistic parallel simulation. In *Proceedings of the Seventeenth Workshop on Parallel and Distributed Simulation*, PADS '03, 2003.

[27] G. Yaun, C. D. Carothers, and S. Kalyanaraman. Large-scale tcp models using optimistic parallel simulation. In *Workshop on Parallel and Distributed Simulation*, pages 153–, 2003.

[28] G. R. Yaun, D. Bauer, H. L. Bhutada, C. D. Carothers, M. Yuksel, and S. Kalyanaraman. Large-scale network simulation techniques: Examples of tcp and ospf models. *SIGCOMM Comput. Commun. Rev.*, 33(3), July 2003.