ABSTRACT

LUO, XIAOQING. ScalaIOExtrap: Elastic I/O Tracing and Extrapolation. (Under the direction of Dr. Frank Mueller.)

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 scalable and elastic traces. After gathering a set of smaller traces, we extrapolate the application trace to a large numbers of nodes, which can even be larger in the number of nodes than any existing supercomputer. We conducted our experiments on a local cluster, an HPC machine of the Argonne Leadership Computing Facility (ALCF) and via the Co-design of the Exascale Storage system (CODES) simulator [9].

The main contributions of this work are the techniques to (a) gather a set of lossless and scalable trace files, (b) build a mathematical model to analyze trace data and extrapolate it to large sizes, and (c) replay the extrapolated trace file to verify its accuracy.

ScalaIOExtrap: Elastic I/O Tracing and Extrapolation

by
Xiaoqing Luo

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina

2015

APPROVED BY:

Dr. Nagiza Samatova	Dr. Kemafor Anyanwu Ogan

Dr. Frank Mueller
Chair of Advisory Committee

BIOGRAPHY

Xiaoqing Luo was born in Hebei, China. She received the B.E degree in Electronic & Information Engineering form South China University of Technology, China, in 2002, the M.S. degree in Communication & Information System from South China University of Technology, China, in 2006. She is currently a MS student in Computer Science department in North Carolina State University. Her research interests cover elastic I/O tracing, I/O extrapolation and I/O replay on High Performance Computing.

ACKNOWLEDGEMENTS

First, I wish to express my sincere gratitude to my supervisor, Dr. Frank Mueller, for his patience, enthusiasm and encouragement. His logical way of thinking and his knowledge have been of great value for me in these two years. I could not have imagined having a better advisor for my MS's study. Besides my advisor, I would like to thank my committee members: Dr. Nagiza Samatova and Dr. Kemafor Anyanwu Ogan for their insightful comments and encouragement. I own my deep gratitude to Phil Carns, John Jenkins, Shane Snyder and Rob Ross from Argonne National Laboratory for their valuable advice and friendly help for my work. I also want to thank my lab-mates for all of their help in these years. Last but not least, my deepest gratitude goes to my parents and my husband for their unflagging love and support throughout my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

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 / map-reduce relying on a parallel file system (PFS). I/O presents a challenge due to complex interactions of multiple software components [7]. Understanding inefficiencies and determining bottlenecks in I/O are thus imperative, and are facilitated by tracing and analyzing I/O performance of parallel applications. However, I/O analysis in parallel systems is not trivial due to multiple I/O layers [19] 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 [14, 22], 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 tracing tool is significant, especially for a large-scale production HPC system [21] (e.g., long application execution time and large number of nodes participating).

- **Static I/O analysis:** Gather the trace information during the 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.

I/O tracing can also be performed by modeling and predicting applications' behavior [6]. Unfor-

tunately, such an approach can only provide overall statistics for an application on a particular architecture, and may not satisfy the needs for detailed analysis.

1.1 Hypothesis

We hypothesize that I/O behavior of parallel applications can be analyzed from a set of traces gathered from executions with smaller number of ranks and extrapolated to a trace of larger number of ranks while retaining correct communication patterns and execution times for mesh-based communication patterns.

1.2 Contribution

In this thesis, we contribute ScalaIOExtrap, a novel tool to provide large-scale traces suitable for exascale I/O analysis. It obtains the lossless I/O access behavior of an application running in a large-scale system without requiring source code. Figure 1.1 gives an overview of ScalaIOExtrap, where RS (Rank Size) defines the number of ranks of a job's communicator obtained from MPI_Comm_size. The main ideas of ScalaIOExtrap are: **(1) Gather a set of lossless and**

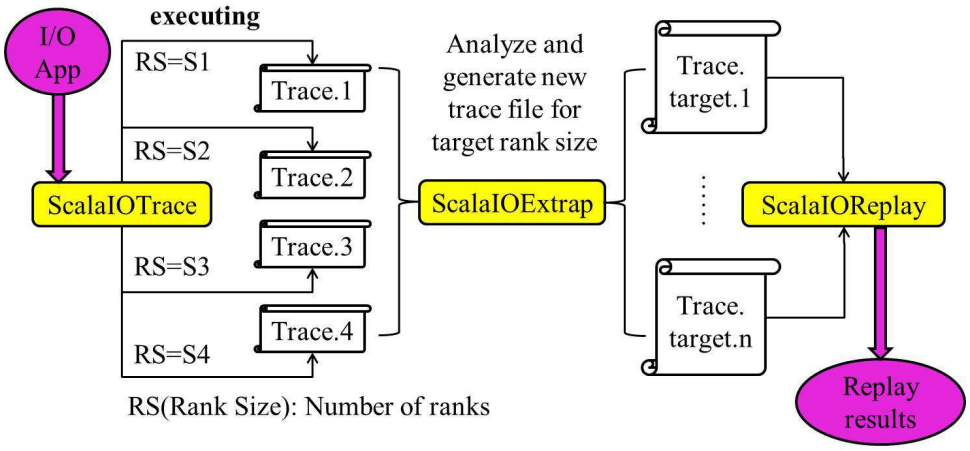

Figure 1.1: Framework of ScalaIOTrace,ScalaIOExtrap and ScalaIOReplay

scalable I/O trace files in a relatively small system: We build **ScalaIOTrace** to achieve this goal. To this end, we extend ScalaTrace [14] to trace both higher level MPI-IO and lower level POSIX-IO operations losslessly. We contribute a number of algorithms to compress the traced events in the trace file, which then becomes nearly constant in size. We consolidate the

events by combining intra-node and inter-node compressions. **(2) Analyze the set of trace files and extrapolate small files into large size trace files: ScalaIOExtrap** is designed to utilize a set of smaller size trace files as input, to extract the corresponding traced data and build a mathematical model to analyze the relationship between data and the number of nodes. We also exploit modeling techniques to analyze the relationship between data itself. **(3) Calculate the extrapolated data and generate a single trace file:** After modeling and analysis, the equation which uses the number of ranks as an input and uses extrapolated data as output is determined. From the extrapolated data we generate a single constant size trace file. **(4) Extend ScalaTrace replay engine to verify the correctness (ScalaIOReplay):** In order to achieve *scalability*, we heavily compress the lossless traced information into a nearly constant size file. We extend the ScalaReplay engine to automatically parse and replay the I/O trace file and its extrapolated I/O calls.

For verifying the accuracy and portability of our approach, we conducted experiments on three different HPC platforms in terms of cluster types: (1) A small-scale cluster, (2) a large-scale cluster, (3) and a virtual cluster (CODES simulator). We discuss each of them in Section 4. 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. Meanwhile, traditional dynamic I/O analysis tools (e.g., ScalaIOTrace) require 8192 processors and more than 200 seconds to gather the same information.

Chapter 2

Background

ScalaIOTrace is designed on our prior work on MPI tracing via ScalaTrace V2 [22]. Similarly, we reuse some techniques of ScalaExtrap [21] for developing ScalaIOExtrap. In this section, we briefly introduce these tools and CODES, which we utilize to verify our work.

2.1 ScalaTrace

ScalaTrace is an MPI communication tracing framework for parallel applications [13]. 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 [15].

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 [14]. 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 file is a single file of nearly constant size with sufficient information to capture all tasks.

Parameters of I/O events are captured as elastic data element representations in ScalaTrace V2 [22], 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 [15, 12, 23, 16]. 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.

2.2 ScalaExtrap [21]

ScalaExtrap 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. In order 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. 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:

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: It 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.

2.3 CODES [9, 10]

The Co-design of Exascale Storage System (CODES) framework is designed for evaluating exascale storage system design points. The CODES framework explores the Rensselaer Optimistic Simulation System (ROSS), which is a parallel discrete-event simulation framework allowing simulations to be run in parallel [26, 24, 1]. Using CODES and ROSS, the ALCF computing environment with the PVFS storage system and the I/O subsystem of the Intrepid IBM Blue Gene/P (BG/P) system of the ALCF can be modeled.

CODES models the 1) network behavior, 2) hardware components and 3) software protocols of the ALCF computing environment. The CODES storage system simulator supports the necessary protocol to handle application-level file open, close, read and write using ALCF hardware and software models.

The CODES simulator also includes a burst buffer tier to minimize the time spent on I/O. The main idea of the burst buffer includes: 1) Reserve space in the burst buffer for application data. 2) Receive the application data and deposit this data into the burst buffer. 3) Once data is buffered, the application will be notified of completion, so that software may then transfer the buffered data to the file system.

Chapter 3

Design And Implementation

I/O analysis is a challenge to multi-layer I/O stacks and multiple I/O patterns in the program. In this section, we introduce (a) capabilities for trace compression, (b) analysis of the trace and extrapolation into target sizes of nodes, (c) replay capabilities on elastic data representations of MPI-IO, POSIX-compliant and UNIX-style I/O at the library level ultimately resulting in operating system (OS) I/O system calls (syscalls). In contrast to MPI communication tracing and past work on extrapolation, we propose a number of novel tracing techniques necessitated by the unique characteristices of parallel I/O.

We design the *ScalaIOTrace*, *ScalaIOExtrap* and *ScalaIOReplay* tools suitable for single program multiple data (SPMD) programs. 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.

3.1 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 and UNIX/C library 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.

3.1.1 Elastic strings

ScalaIOTrace introduces a novel domain-specific compression technique for string elasticity, which is motivated by the filenames used in the three I/O patterns mentioned in Section 1. After inter-compression, filenames generated in all the nodes need to be compressed into one single string for constant trace size representation. For patterns A (Serial I/O) and C (Parallel I/O, shared-file) of a single file, only one filename will be generated, while pattern B (Parallel I/O with one file per process,N-to-N) requires N filenames. We focus on pattern B because patterns A and C are trivial in terms of filenames.

In pattern B, filenames are normally distinguished by rank numbers. Instead of just conventional string recording, filenames are decomposed during inter-node compression to separate rank-specific from rank-agnostic substrings. For example, "rank_10" and "rank_11" may be filenames on ranks 10 and 11, respectively. We establish three preconditions for string compression: (a) Non-numeric substrings have to match. (b) The number of numeric substrings is the same. (c) Numeric substrings start at the same relative index. We strictly enforce these preconditions by extracting the numeric part and recording its relative index. There maybe multiple numbers in one filename. We denote identical numbers once and compress the rank-specific numbers as a RSD pattern. Consider the generated filename "$/dir0/file\_ < rank >$ " corresponding to tracing and compressing results shown in Figure 3.1. If the rank-specific part is an arith-

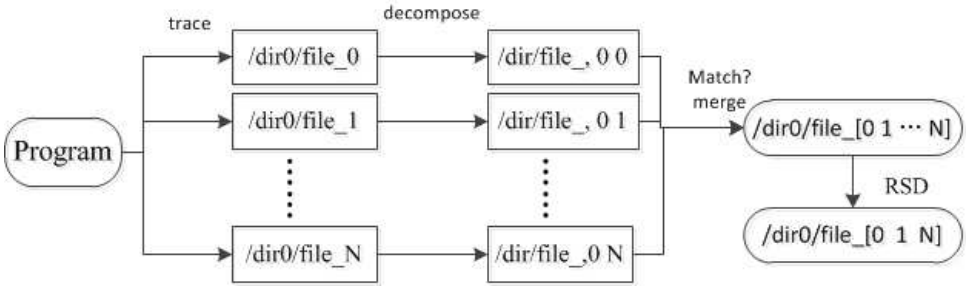

Figure 3.1: Tracing and RSD merge of filename

metic progression, which is most common in HPC programs, we compress it as a RSD pattern [$start\ stride\ size$], i.e, $/dir0/file\_$[0 1 N] start=0, stride=1, and size=N.

3.1.2 Elastic data element representation

Based on prior work of data element representation [22], we utilize compression on the relation between data elements. In collective I/O events, tracing and compression data parameters such

as $OFFSET$ and $COUNT$ are very important. For example, in parallel collective I/O, different ranks have a different $OFFSET$, so they cannot be merged into a single form if traces shall be lossless nor can they be multiple items if traces shall scale.

We propose another level of compression beyond prior work. During inter-compression, we compare the differences between adjacent ranks. If they are an arithmetic progression, we merge them as RSD pattern $< start\ stride\ size >$; otherwise, each rank generates one offset (recorded in preorder traversal of a binary tree). Example: The offset of a collective I/O call running on 4 nodes is calculated as $rank * c$, where c is a constant. After inter-node compression, we record it as: $[-1\ 0\ c\ 4]$. If the offset cannot be merged into a RSD pattern, it is recorded as: $[-2\ offset0\ offset1\ offset3\ offset2]$. -1 is used to indicate a RSD pattern and -2 to indicate a regular item.

To the best of our knowledge, most collective I/O calls have offsets with an arithmetic progression, so this approach can best provide scalability and also facilitate extrapolation. Due to intra-node compression, there are chances that each rank may generate multiple offsets (see extrapolation section).

3.1.3 Elastic handles

MPI file handles for file objects and the MPI_Datatype for a derived data type are opaque pointers maintained by MPI on a per-task basis. They are meaningless for extrapolation and replay. Instead of storing the handle, it is encoded as an integer index and subsequently decoded during replay, where it is associated with the pointer obtained by issuing the respective call. To make it identical, commonly used MPI datatypes such as MPI_INT are also encoded as an integer index and will be decoded as the corresponding type during replay. We restrict the commonly used MPI datatype buffer size to 1000. For newly generated MPI datatypes, append their value to the buffer after index 1000.

There are other opaque pointers, such as request and status handles, which are treated in the same manner as indices into type specific vectors, i.e., their pointers/values are populated dynamically during replay.

3.2 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 characters, e.g., for string-based parameters such as filenames and data-based parameters such as offsets. ScalaTrace is a looseless 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, assuming that they have the same number of events.

3.2.1 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_0th event is MPI_File_open for input trace files, then we also generate an MPI_File_open as the n_0th event for the target trace file. ScalaTrace records rank lists at the event level. We exploit Gaussian Elimination introduced in our prior work [21] to extrapolate these ranklists for mesh/stencil patterns.

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 loop iteration times of each event. For *weak scaling* (where the workload assigned to each processor stays constant as number of processor 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., workload assign to each processor decreases as number of processor increases) and also for tracing the lower level POSIX-IO for collective MPI-IO calls [3], 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 the factors and calculate the loop iterations for a target number of ranks.

3.2.2 Elastic string extrapolation

Extrapolation for strings, especially filenames, is important in ScalaIOExtrap. Filenames plays major role in distinguishing different I/O patterns (see section 1). For pattern A (Serial I/O) and C (Parallel I/O, shared-file), merged 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 "spokesman" 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 *start*, *stride*, *size* at RSD pattern have linear relationships to rank size. In most cases, the *start and stride* does not change, only *size* changes with rank size. We simply generate the equations over the reference of traces and solve them using Gaussian Elimination.

Table 3.1: 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

3.2.3 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 3.1 after compression.

```
MPI_Offset offset = rank*(960/rank_size);
MPI_File_seek(... offset ...);
```

For the column dimension, the values of each column in Table 3.1 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. If we extrapolated to 8 ranks, we cannot just extrapolate in one dimension: (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. We create a mathematical model for column extrapolation using the four models introduced in Section 2 plus a new model:

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

For the data extrapolation, we do not use the smallest standard deviation to decide which model is the correct one, because we want to get exact results, not an approximate one. So as long as the standard deviation is not zero, we assume that we cannot extrapolate. We create the most common models. We also provide an interface for users to add their own models for extrapolation.

We extrapolate the column dimension considering 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+b model to predict the results for a target rank size. After obtaining the first parameters from column extrapolation, we combine them with the equation from row extrapolation and then calculate the remaining parameters.

3.2.4 Handlers and time extrapolation

As mentioned in Section 3.1, handlers are coded into integers. We just use the same technique as for extrapolating data-based parameters. Normally, handler extrapolation is simple. E.g., for a file handle, in either SIO, N-to-N or N-to-1 I/O pattern, all ranks perform same file open operation regardless of rank size, which remains unchanged during extrapolation. For strong scaling, the open operation depends on rank size. We can also handle this with the techniques mentioned above. We reuse time extrapolation of our prior work by mathematical modeling.

3.3 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, 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. We focus on the unique aspects for I/O replay in the following.

3.3.1 Elastic string replay

Let us use filenames as an example for elastic string replay (other string replays are straightforward, e.g., open mode in fopen). The filename of the SIO and N-to-1 I/O patterns can simply be extracted from the traces. Let us focus on how to replay filenames for the N-to-N I/O pattern.

Collections of strings and numeric representations are replayed by decomposition and recomposition to generate parameters to issue I/O calls. The compressed filename "$/dir0/file\_[0\ 1\ N]$" is replayed as follows: (a) Extract numerical substrings (here: $[0\ 1\ N]$). (b) Decide the start, stride and size in RSD numerical substrings (here: $start = 0, stride = 1, size = N$). (c) Use the equation $start + stride * rank\_numer$ to obtain the corresponding number (here: rank one

will get $0 + 1 \times 1 = 1$). (d) Recompose filename (here: rank one recompose its filename as $/dir0/file\_1$).

3.3.2 Elastic file handle replay

File handles are represented as integer indices after encoding. We generate a map container during handle replay using the index as a key corresponding to the filename and a pointer handle. Upon encountering events with handles, a new tuple $< index, (filename, filehandle) >$ is generated per actual file handle. The actual handle can thus be uniquely identified and associated with respective replayed events. When a handle is deallocated (at a file close), the tuple is removed from the map container.

3.3.3 Elastic datatype replay

MPI_Datatype parameters are expanded during replay as follows: (a) Generate a buffer and reserve entries for default MPI datatypes according to the original encoding. (b) Relegate the datatype to the buffer. (c) Dereference the item at the respective encoded offset in the buffer. When there is a new MPI_Datatype generated, we also push it into the buffer to the position where it was recorded, so that it can be used later by other calls.

3.3.4 Time-accurate and semantically correct replay

Event replays may be motivated by different goals with respect to timing. (1) The objective is to preserve I/O timings of the original application. This might be motivated by performing root-cause analysis of I/O problems. (2) Another objective is to determine how fast the application might run on another HPC system and/or filesystem. This could be motivated by testing communication events on different network interconnects and/or different file systems, I/O nodes and I/O server compositions. In both cases, such analysis and execution becomes feasible by replaying from the actual trace file, i.e., without actually porting the application to the new target, only the replay engine needs to be ported once for any set of application traces. Depending on replay objectives, ScalaIOReplay lets users choose one of these modes using environment variables.

3.3.5 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. 3.2) between ScalaIOReplay and CODES to convert the ScalaTrace data structure to the CODES data structure. CODES is a POSIX-IO-based simulator. To preserve

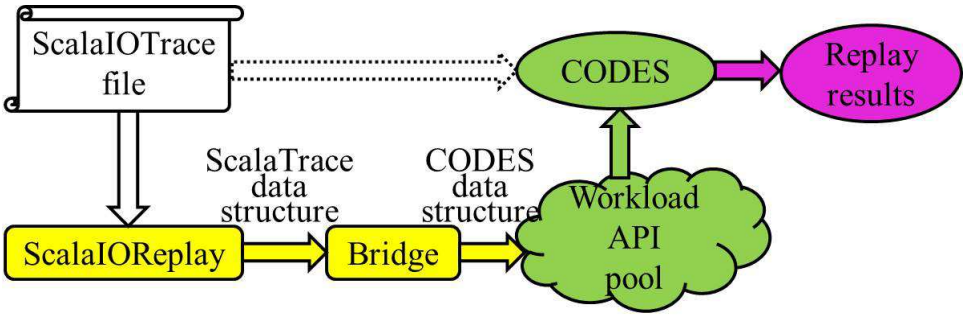

Figure 3.2: Bridge between ScalaIOReplay and CODES

consistency, we ignore the higher level in the I/O stack [19] 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 an 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.

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. Bridge processing 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 1. 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 data

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. Instead, 16 different aggregators write the 4096*1024 bytes in parallel. 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 of MPI on ALCF by default. Whenever the I/O volume exceeds max-blocksize, we separate a single I/O call into multiple I/O operations.

Chapter 4

Experimental Framework

As mentioned in Section 1, to verify the correctness of our approach, we deployed our framework on top of: (1) ARC (x86_64): **A** **R**oot **C**luster for Research into Scalable Computer Systems at North Carolina State University with 1728 cores on 108 compute nodes. It supports the Network File System (NFS), Parallel Virtual File System (PVFS2) and a local filesystem. (2) ALCF (Blue Gene/Q): The **A**rgonne **L**eadership **C**omputing **F**acility, a large HPC center that provides a large, parallel storage system shared between multiple HPC resources [9]. It supports the: IBM General Parallel File System (GPFS). (3) CODES (Simulator): Co-design of Exascale Storage system [9, 10]. It supports the Parallel Virtual File System (PVFS/BGP) [9]. We evaluate the correctness of our approach for the three aspects shown in Figure 4.2, Figure 4.1 and Figure 4.3:

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 same. However, the delta execution time we extrapolated will have some variance. So we ignore the execution time and compare the structure of the two trace files.

C2 We replay the extrapolated trace file and compare the execution time with the original program running on same number of processors. The two execution times should be similar.

C3 We use **Darshan** tracing [3, 2] 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 involved by 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.

4.1 ARC

We first conduct experiments on a local 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 depends 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). Figure 4.1 depicts the set of 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).

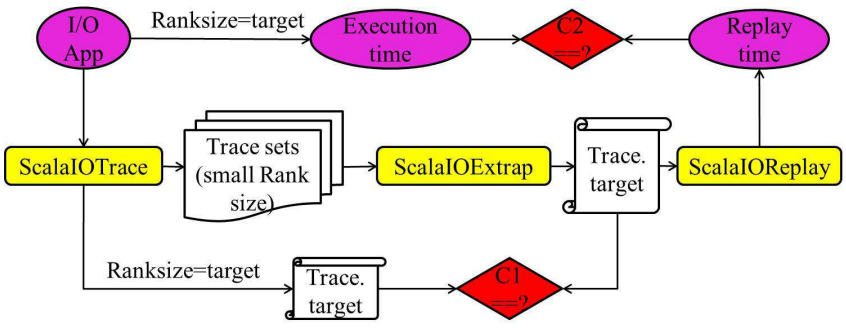

Figure 4.1: Extrapolation Verification on ARC

4.2 ALCF

We also conduct experiments on ALCF (Cetus) with the configuration shown in TABLE 4.1. The Darshan tracing tool, which records statistics such as the number of files opened, time spent

Table 4.1: Configuration of ALCF (Cetus)

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	Mmeory/core: 1 GB

in performing I/O, and the amount of data accessed by an application, is natively supported on ALCF (Cetus). We also analyze Darshan logs by using the darshan-job-summary.pl utility. We show the verification experiments for ALCF (Cetus) in Fig. 4.2.

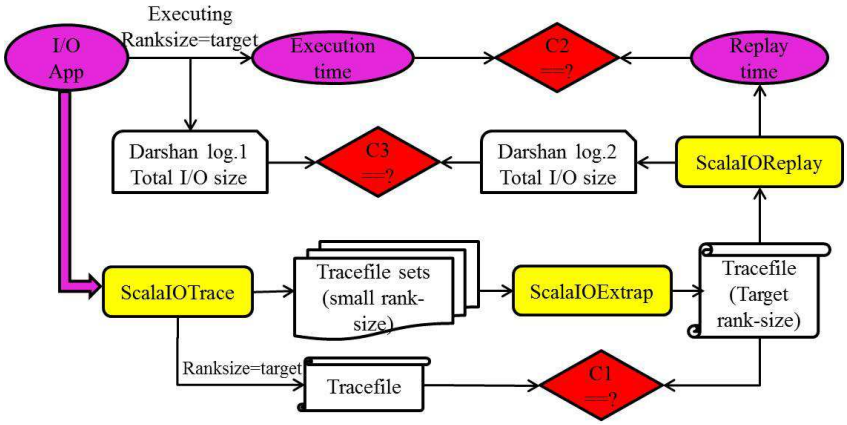

Figure 4.2: Extrapolation Verification on ALCF

4.3 CODES

ROSS [25] offers two methods of simulation at once: the optimistic and the conservative mode. The optimistic mode is a fast mode, where "reverse function handlers" in simulator execute events out of order without violating any timing dependency. The conservative mode does not need "reverse function handlers" and is implemented by CODES. We conduct simulations on

a x86_64 machine, execute events sequentially, 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 Figure 4.3.

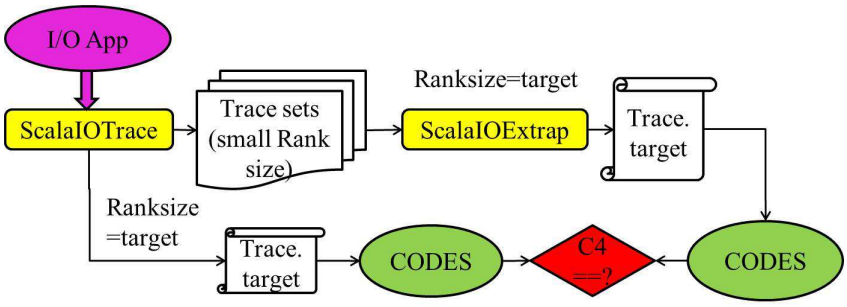

Figure 4.3: Extrapolation Verification on CODES

Chapter 5

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. 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 slight difference between the extrapolated trace replay time and the observed execution time is deemed to exit. The difference in execution time 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.

5.1 Results on ARC

In order to verify the correctness as well as the accuracy of ScalaIOExtrap, we conducted our experiments on various filesystems. As shown in Fig. 4.1, we compare the traces and execution times on ARC with the filesystems: **(1) Local Filesystem** using local data storage devices; **(2) Network Filesystem (NFS)** a shared filesystem that allows a user on a client computer to access files over a network; and **(3) Parallel Virtual File System (PVFS2)** a file system that distributes its data over multiple storage ndoes.

5.1.1 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 as POSIX-IO (I/O size: 8KB per processor; iterations: 100), MPI-IO (iterations: 3; I/O size : 1M, 2M and 3M bytes) 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 "$diff - wi$" to compare the ScalaIOTrace and extrapolated traces with the same number of ranks. Excepts for the time extrapolations, they match perfectly. Timings (y-axis) are shown for different number of ranks (x-axis) in Figure 5.1. For the same I/O size and the same I/O

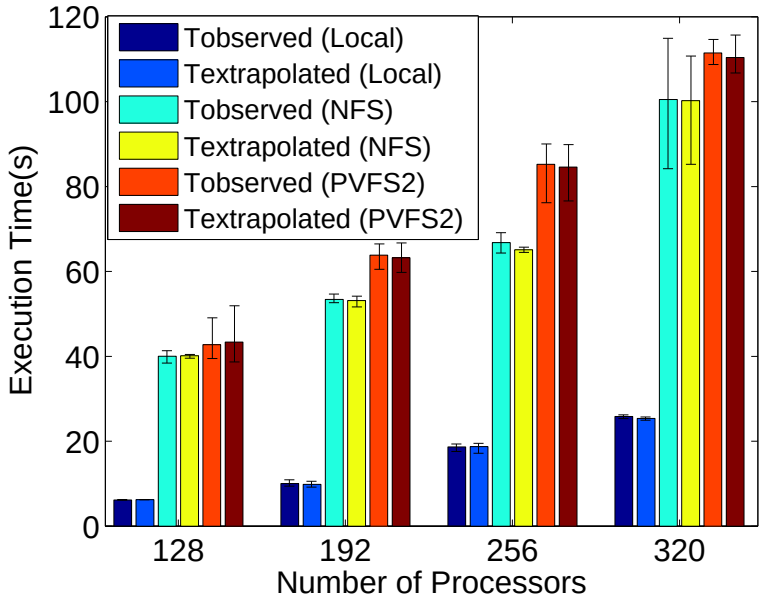

Figure 5.1: Results of IO-sample benchmark in Local, NFS and PVFS2

pattern, the local filesystem takes the shortest time because it is faster for nodes to access their local memory. PVFS2 takes the longest time as explained later for IOR results. The replay time fluctuates with application execution time. Results show that time inaccuracy is within 5%.

5.1.2 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. Both shared-file (chunk pattern) and shared-file (interleaved pattern) 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 [18]. 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 the regions of the file that is being written, 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 "$diff$" 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 Figure 5.2 and Figure 5.3.

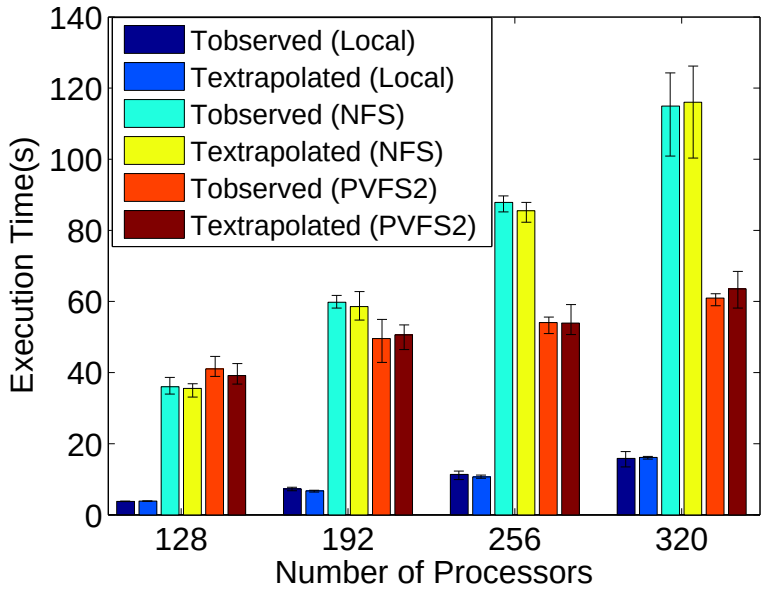

Figure 5.2: Results of IOR (chunk pattern)

Similar to IO-sample, the execution times for the local file system are the shortest among the three file systems. Chunk and interleaved patterns do not differ for local storage, because nodes access private resources.

For the NFS file system, collective buffering of MPI-IO makes the interleaved pattern faster than the chunk pattern because NFS is tuned for few, large operations instead of many small

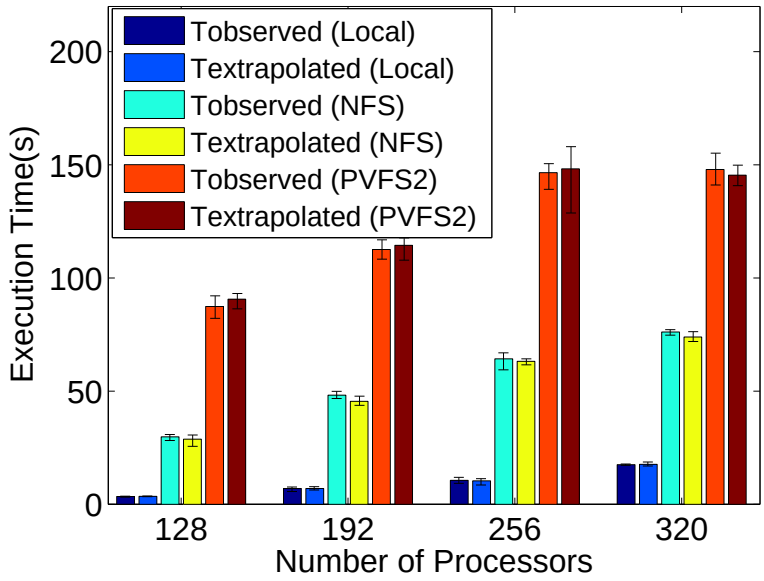

Figure 5.3: Results of IOR (interleaved pattern)

ones.

However, this is not the case for PVFS2. Here, the interleaved pattern takes more time than the chunk pattern, because each file stored on PVFS2 is broken into smaller parts distributed across servers. In most other parallel I/O cases, PVFS2 is efficient because it leverages all available resources evenly and allows the load to be distributed. However, if a single process of an application accesses the file system serially, distributing chunks across all servers increases overhead without gaining any benefit from parallelism [17]. PVFS2 imposes overhead for the interleaved pattern because files are accessed serially, while the chunk pattern becomes more efficient because of its parallel accesses. That is why the chunk pattern outperforms the interleaved pattern for PVFS2, i.e., PVFS2 performs better than NFS for the chunk pattern but worse than NFS for the interleaved pattern. It also explains the IO-sample results mentioned before.

We also obtain timing results per processor for the N-to-N pattern (see Figure 5.4). When comparing the results shown in Figure 5.2, Figure 5.3 and Figure 5.4, 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.

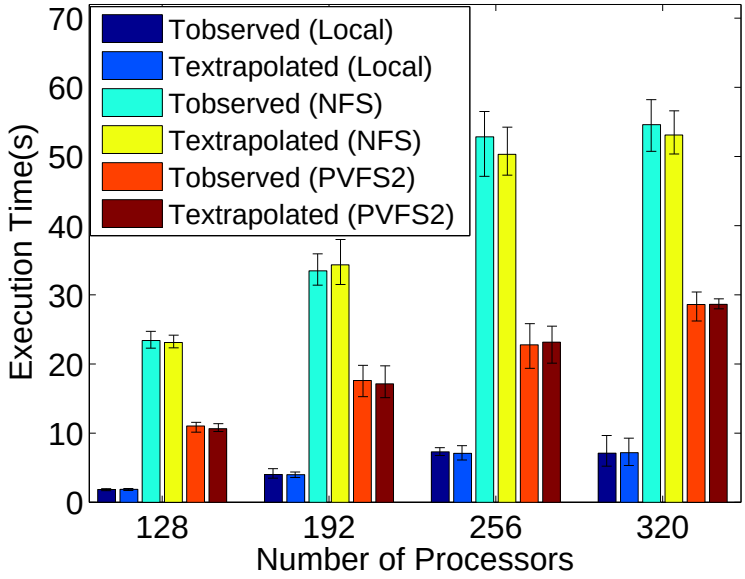

Figure 5.4: Results of IOR file-per-processor

5.2 Results on ALCF

Besides comparing traces and execution time, we also compare the total I/O size on the ALCF system (see Figure 4.2) using the default installed 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 read traces into memory, which introduces overhead proportional to its size. We capture the extrapolated I/O size as: $S_{extrapolated} - (S_{trace} \times RS)$ where $S_{extrapolated}$ 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). We select the input parameters for POSIX-IO (I/O size: 4KB per processor; iteration times: 100), MPI-IO (iteration times: 2; I/O size: 64KB and 128KB) and gather traces from 16, 32, 48 and 64 ranks as the small trace set for extrapolation. Execution time results depicted in Figure 5.5 (on a logarithmic scale on the y-axis) indicate an average time difference of no more than 5%.

We also gather IOR execution time results for shared-file (interleaved), 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 Figure 5.6. ALCF supports 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 Figure 5.6 (again with a logarithmic

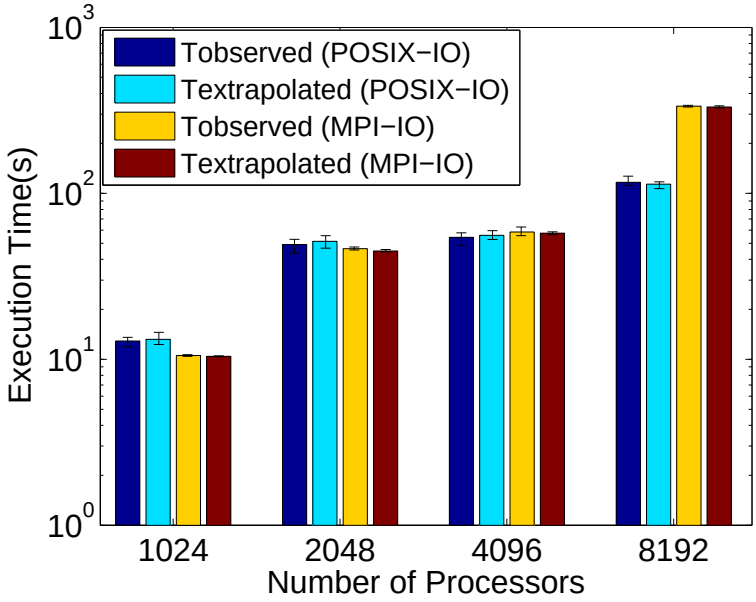

Figure 5.5: Results of IO-sample on ALCF

y-axis). For the same I/O size and with MPI-IO collective buffering (see Section 5.1), 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 bechmarks, we also use the "$diff$" 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.

We assess how closely our ScalaIOExtrap matches the original I/O size using the verification from Figure 4.2.

After eliminating the ScalaIOReplay overhead, results in Figure 5.7 (*Observed* indicates the I/O size of application while *Extrapolated* indicates the I/O size of replaying extrapolated trace) indicate that the total I/O size gathered from ScalaIOReplay are exactly same as the I/O size gathered from an application run.

5.3 Results for CODES

CODES is a virtual HPC simulation system and gives us 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

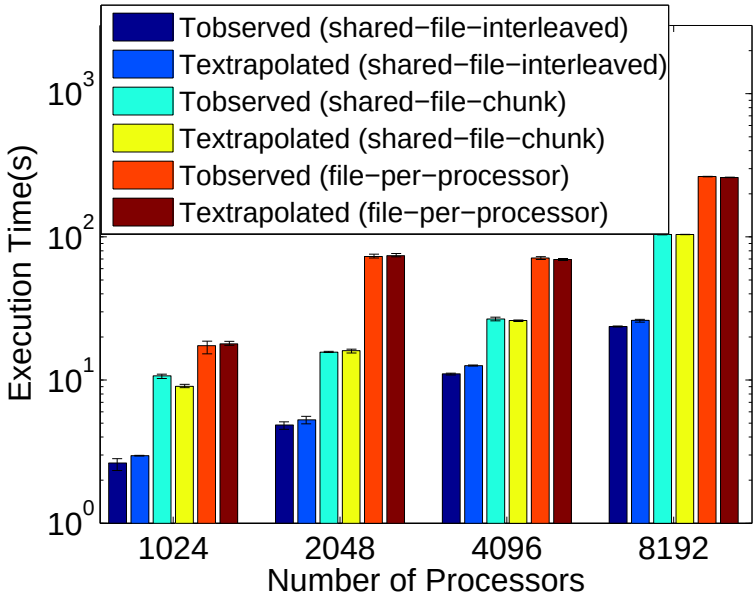

Figure 5.6: Results of IOR on ALCF

as shown in Figure 4.3. Our goal is to verify the correctness of our approach, i.e., although we can extrapolate traces to extremely large numbers of ranks and 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 Figure 5.8 with a logarithmic y-axis). We compare the total number of opens, reads, writes and closes over all processors between the application trace (Observed) and extrapolated trace (Extrapolated). Let N be the number of operations for *Observed* (application) and *Extrapolated* (extrapolated trace replay), e.g., *Observed Nreads* is the total number of reads of the application. Observed and extrapolated results are identical for 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 262144 bytes, we obtain the collective MPI-IO results depicted in Figure 5.9 (logarithmic y-axis). The extrapolated number of I/O operations 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. Figure 5.10 depicts the difference for extrapolations to 512, 1024, 2048 and

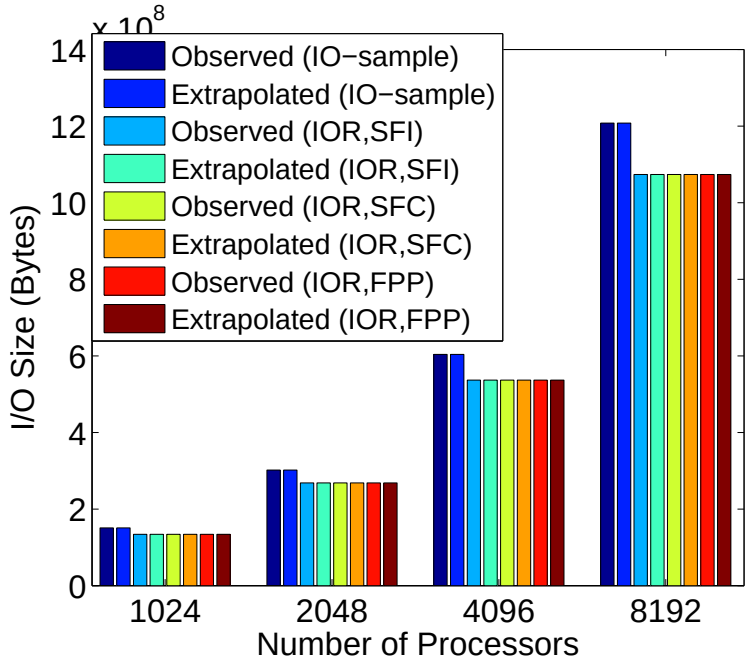

Figure 5.7: I/O size of IO-sample, IOR shared-file-interleaved (SFI), IOR shared-file-chunk (SFC) and IOR file-per-processor (FPP) on ALCF

4096 ranks. The max-blocksize does not effect the number of open and close calls, so we only compare the number of reads and 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 the difference increases as the 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 Figure 5.11, logarithmic y-axis). Results indicate that under the file-per-processor (N-to-N) mode, our approach extrapolates traces correctly.

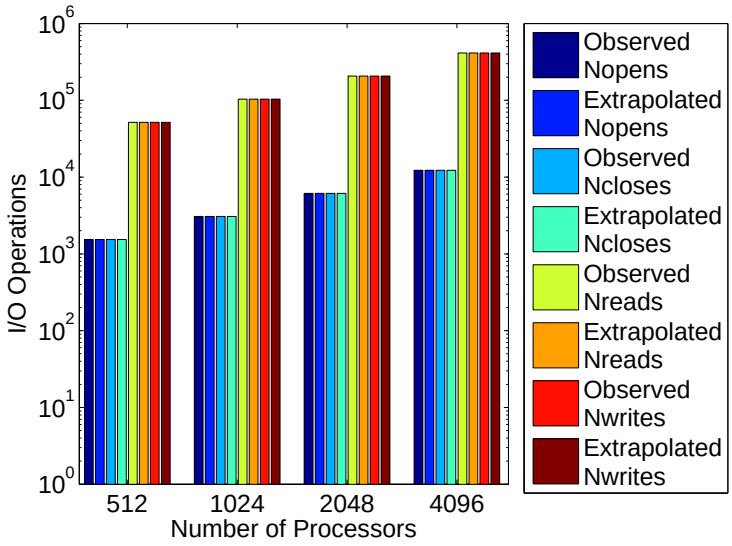

Figure 5.8: I/O operations of IO-sample (POSIX-IO)

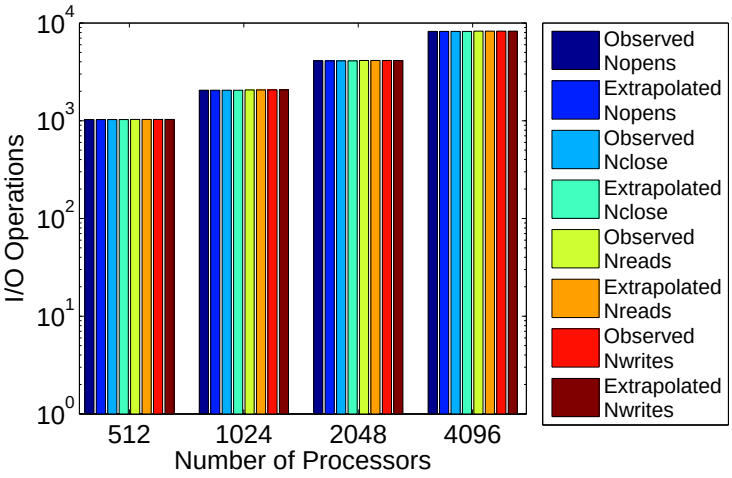

Figure 5.9: I/O operations of IO-sample (MPI collective I/O)

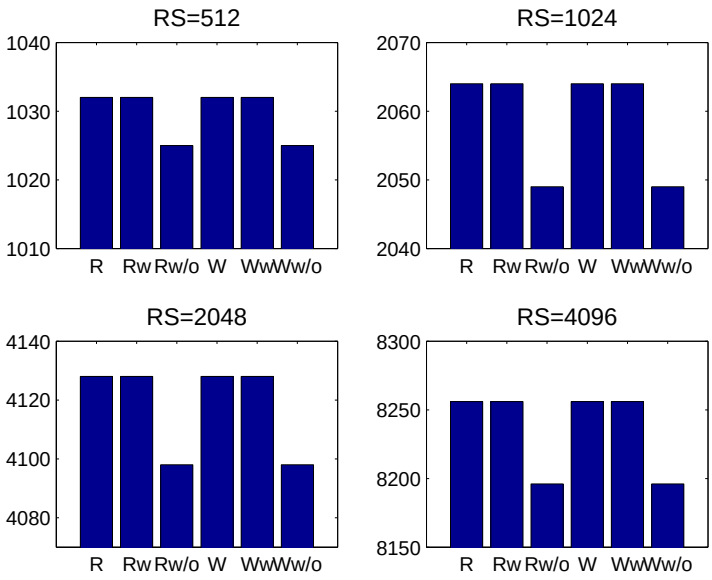

Figure 5.10: I/O operations of with and without max-blocksize (IO-sample, MPI collective I/O)

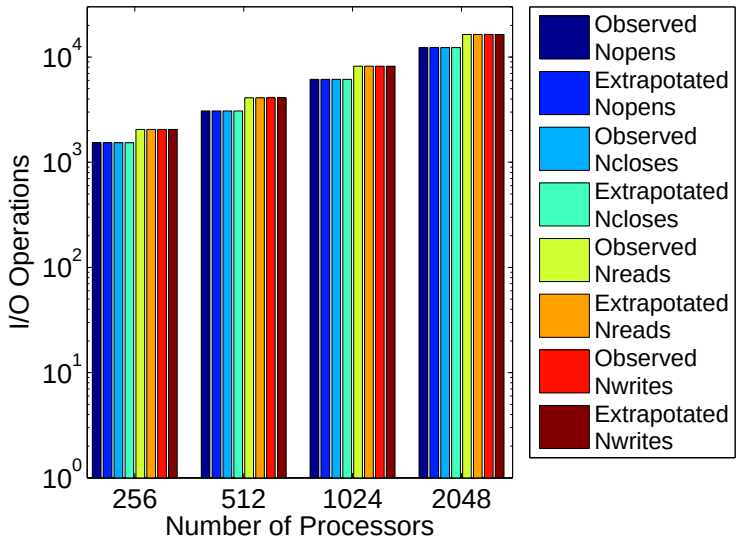

Figure 5.11: I/O operations of IOR (file-per-processor)

Chapter 6

Related work

As I/O is playing an increasingly critical role in high performance computer (HPC) systems, more and more I/O performance analysis techniques have been proposed.

Andrew et al. [8] proposed an analysis framework based on server-side tracing data. By utilizing the trace data collected on large-scale network file system, they investigated the following aspects: (1) changes in file access patterns, (2) properties of file I/O and file sharing, and (3) the relationship between file type and client access patterns. They aimed at optimizing workload management. Yang et al. [11] devised I/O signature identifiers, an approach to characterize per-application I/O behavior on the server-side in forms of the I/O volume read/written by the applications, the frequency of the I/O operations, and throughput achieved on a file system. They exploited a set of statistical techniques to extract the I/O signature by utilizing noisy, zero-overhead server-side I/O throughput logs on leadership-class HPC systems. In contrast, our work focuses on analysis of I/O behavior when executing on a large-scale HPC systems, and our objective is to save time and resources.

Wright and Hammond [20] proposed a technique. In this work, they 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. Furthermore, they studied the read/write time and the time spent obtaining or releasing per-file locks. Also, they analyzed how MPI-IO and low-level POSIX-IO perform for different I/O benchmarks and different rank sizes. Compared to their work, we focus on extrapolating I/O traces to arbitrary number of ranks instead of just analyzing trace data for the existing small scale of ranks. Moreover, we are able to extrapolate accurate I/O size and I/O operations at both the MPI-IO and POSIX-IO layers.

Eckert and Nutt [4, 5] studied the extrapolation of trace data of multiple threaded programs on shared memory multiprocessors. They collected a source trace by observing the execution of a program on a particular architecture. Given a program and data set, they adjust the source

trace without reexecuting to produce a correct trace for a parametrically different execution architecture. In this work, the authors focus on studying multi-threaded shared memory programs, as well as the correctness of extrapolated traces under the existence of non-determinism, which is caused by moving the trace accross different architectures. In contrast, our work focus on I/O traces and is based on deterministic application execution, i.e., we preserve the causal orders both for ScalaIOTrace and ScalaIOExtrap.

Chapter 7

Conclusion and future work

We presented the design and implementation of three tools: **(1) ScalaIOTrace** is a multi-level I/O tracing approach, which supports both the MPI-IO and POSIX-IO interpositioning. It utilizes singletons, vectors and RSDs in an elastic trace representation. Tracing results 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 are retained perfect accuracy, and execution time remains sufficiently accurate.

(3) ScalaIOReplay is a parsing and replay engine, which re-executes the events of ScalaIO-Trace traces in the same order as original applications. 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 replay the extrapolated trace for large number of ranks by collecting a set of traces from with small number of ranks. We preserve event ordering and time accuracy in these large traces. With this technique, large-scale communication 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.

The next step of our work will be extending ScalaIOExtrap to extrapolate non-identical trace patterns and data-driven parameters such as offsets. Now we assume all four input traces have

the same pattern (same numbers, and same names for corresponding event), however, there maybe complex applications which generate different trace patterns for different number of ranks. Also, we assume all the data-driven parameters and timing information meet the models we already built, however, more models need to be considered. Our intent is to generally and correctly extrapolate for more complex applications.

REFERENCES

[1] David W. Bauer Jr., Christopher D. Carothers, and Akintayo 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.

[2] 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.

[3] Philip H. Carns, R. Latham, Robert B. Ross, K. Iskra, Samuel 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.

[4] Zulah Karen Fields Eckert. Trace extrapolation for parallel programs on shared-memory multiprocessors. *Technical Report, Spring 5-1-1996*, 1995.

[5] Zulah KF Eckert and Gary 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] 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.

[7] Seong Jo Kim, Yuanrui Zhang, Seung Woo Son, Ramya Prabhakar, Mahmut Kandemir, Christina Patrick, Wei-keng Liao, and Alok Choudhary. Automated tracing of i/o stack. In *EuroMPI*, 2010.

[8] Andrew W. Leung, Shankar Pasupathy, Garth Goodson, and Ethan L. Miller. Measurement and analysis of large-scale network file system workloads. In *USENIX 2008 Annual Technical Conference*, ATC'08, 2008.

[9] Ning Liu, Christopher Carothers, Jason Cope, Philip Carns, Robert Ross, Adam Crume, and Carlos 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.

[10] Ning Liu, Jason Cope, Philip Carns, Christopher Carothers, Robert Ross, Gary Grider, Adam Crume, and Carlos 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.

[11] Yang Liu, Raghul Gunasekaran, Xiaosong Ma, and Sudharshan S Vazhkudai. Automatic identification of application i/o signatures from noisy server-side traces. In *FAST*, pages 213–228, 2014.

[12] Frank Mueller, Xing Wu, Martin Schulz, Bronis R De Supinski, and Todd Gamblin. Scalatrace: tracing, analysis and modeling of hpc codes at scale. In *Applied Parallel and Scientific Computing*, pages 410–418. Springer, 2012.

[13] Michael Noeth, Frank Mueller, Martin Schulz, and Bronis R De Supinski. Scalable compression and replay of communication traces in massively p arallel e nvironments. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–11. IEEE, 2007.

[14] Michael Noeth, Prasun Ratn, Frank Mueller, Martin Schulz, and Bronis 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.

[15] Prasun Ratn, Frank Mueller, Bronis R de Supinski, and Martin Schulz. Preserving time in large-scale communication traces. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 46–55. ACM, 2008.

[16] Daniel A Reed, PC Roth, Ruth A Aydt, KA Shields, LF Tavera, RJ Noe, and BW Schwartz. Scalable performance analysis: The pablo performance analysis environment. In *Scalable Parallel Libraries Conference, 1993., Proceedings of the*, pages 104–113. IEEE, 1993.

[17] PVFS Development Team. Pvfs tuning. http://www.pvfs.org/cvs/HEAD-docs/doc/pvfs2-tuning/.

[18] Rajeev Thakur, William Gropp, and Ewing Lusk. Optimizing noncontiguous accesses in mpi – io. *Parallel Comput.*, 28(1):83–105, January 2002.

[19] Karthik Vijayakumar, Frank Mueller, Xiaosong Ma, and Philip C. Roth. Scalable i/o tracing and analysis. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, PDSW '09, 2009.

[20] Steven A Wright, Simon D Hammond, Simon J Pennycook, and Stephen A Jarvis. Light-weight parallel i/o analysis at scale. In *Computer Performance Engineering*, pages 235–249. Springer, 2011.

[21] Xing Wu and Frank Mueller. Scalaextrap: Trace-based communication extrapolation for spmd programs. *SIGPLAN Not.*, 46(8), February 2011.

[22] Xing Wu and Frank 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.

[23] Xing Wu, Karthik Vijayakumar, Frank Mueller, Xiaosong Ma, and Philip 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.

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

[25] Garrett Yaun, Christopher D. Carothers, and Shivkumar Kalyanaraman. Large-scale tcp models using optimistic parallel simulation. In *Proceedings of the Seventeenth Workshop on Parallel and Distributed Simulation*, PADS '03, pages 153–, Washington, DC, USA, 2003. IEEE Computer Society.

[26] Garrett R. Yaun, David Bauer, Harshad L. Bhutada, Christopher D. Carothers, Murat Yuksel, and Shivkumar Kalyanaraman. Large-scale network simulation techniques: Examples of tcp and ospf models. *SIGCOMM Comput. Commun. Rev.*, 33(3), July 2003.