HPC I/O Trace Extrapolation [*]

Xiaoqing Luo[1], Frank Mueller[1],
Philip Carns[2], John Jenkins[2], Robert Latham[2], Robert Ross[2], Shane Snyder[2]

[1] Department of Computer Science, North Carolina State University, Raleigh, NC
[2] Mathematics & Computer Science Division, Argonne National Laboratory, Argonne, IL 60439

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 elastic traces. After gathering a set of smaller traces, we extrapolate the application trace to a large numbers of nodes. The traces can in principle be extrapolated even beyond the scale of present-day systems. Experiments with I/O benchmarks on up to 320 processors indicate that extrapolated I/O trace replays closely resemble the I/O behavior of equivalent applications.

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 applications that rely on parallel file systems (PFSs). I/O presents a challenge due to complex interactions of multiple software components [5]. 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 [17] 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 [12, 20], 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 [19] (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 [4]. 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.

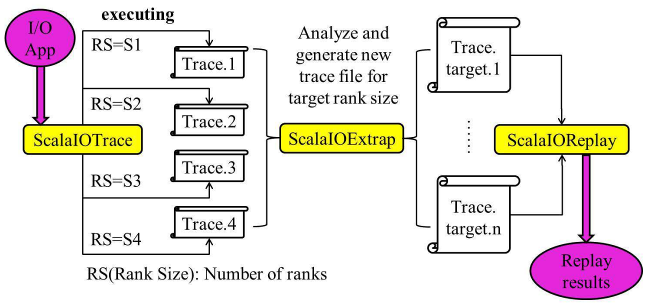

Figure 1: Framework of ScalaIOTrace,ScalaIOExtrap and ScalaIOReplay

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.

[*]This work was supported in part by NSF grants 0958311, 1058779, 1217748 and by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357. This research used resources of the Argonne Leadership Computing Facility.

ESPT2015, November 15-20, 2015, Austin, TX, USA
© 2015 ACM. ISBN 978-1-4503-3997-1/15/11 ...$15.00
http://dx.doi.org/10.1145/2832106.2832108

Contributions of ScalaIOExtrap:

In this work, we augment the existing ScalaIOTrace tracing tool to capture additional information (including POSIX I/O operations) necessary for extrapolation. We then propose a set of novel I/O extrapolation techniques to account for strong and weak scaling, N-1, N-M and N-N file access models, and a variety of access patterns. Finally, we demonstrate the effectiveness of our techniques by implementing an extrapolation tool (ScalaIOExtrap) and applying it to representative HPC I/O workloads.

We conducted experiments on a real-world HPC cluster to verify the accuracy and portability of our approach. The results indicate that the extrapolated trace file captured exactly the same behavior as performed by the I/O application.

2. BACKGROUND

ScalaIOTrace is designed on our prior work on MPI tracing via ScalaTrace V2 [20]. Similarly, we reuse some techniques of ScalaExtrap [19] for developing ScalaIOExtrap.

ScalaTrace is an MPI communication tracing framework for parallel applications [11]. 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 [13].

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 [12]. Nested loops become power-RSDs (PRSDs), i.e., recursively structured RSDs.

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 [20], 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 [13, 10, 21, 14]. 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[19] 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 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.

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 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. 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 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.

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 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, 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 [19] 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 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., the workload assigned to each processor decreases as number of processor increases) and also for tracing the lower level POSIX-IO for collective MPI-IO calls [1], 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 "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 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 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 1 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 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 Handles and time extrapolation

As mentioned in Section 3.1, handles are coded into integers. We just use the same technique as for extrapolating data-based parameters. Normally, handle 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 address 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, it 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.

4. EXPERIMENTAL FRAMEWORK

We evaluate the correctness of our approach as follows:

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.

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 [15]. 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.

We conduct experiments on a 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). Fig. 2 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).

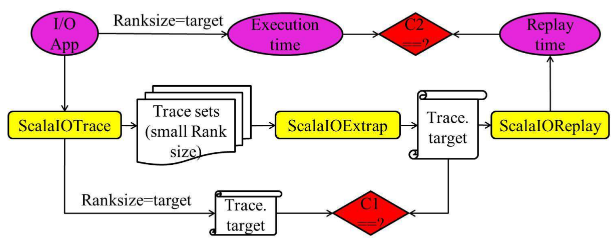

Figure 2: Extrapolation Verification on ARC

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 and operating system noise. We therefore anticipate a slight difference between the extrapolated trace replay time and observed execution time in all cases. 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. 2, we compare the traces and execution times on ARC with the available filesystems (local, NFS, PVFS2).

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 Fig. 3. For the

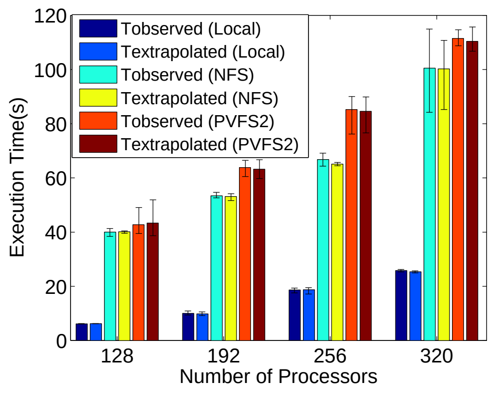

Figure 3: Results of IO-sample benchmark in Local, NFS and PVFS2

same I/O size and the same 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 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. 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 [16]. 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, except for local, which is parallel/N-to-N) in Fig. 4 and Fig. 5.

Similar to IO-sample, the execution times for the local file sys-

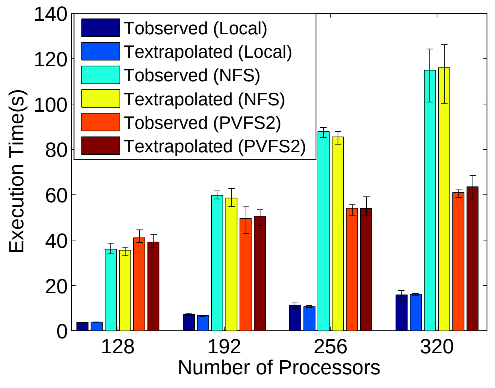

Figure 4: Results of IOR (chunk pattern)

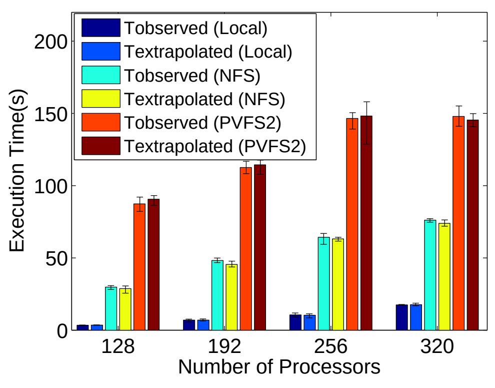

Figure 5: Results of IOR (interleaved pattern)

tem 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 ones. We anticipated that collective buffering would benefit PVFS2 as well [6], but this is not the case in Fig. 5. This may be due to a poor interaction between the default MPI-IO and PVFS2 tuning parameters on this platform; we will investigate this phenomenon further in future work. For the purposes of this research, we observe that both the original application and the extrapolated trace produce the same result.

We also obtain timing results per processor for the N-to-N pattern (see Fig. 6). When comparing the results shown in Fig. 4, Fig. 5 and Fig. 6, 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.

6. RELATED WORK

Leung et al. [7] proposed an analysis framework based on server-side tracing data. Liu et al. [8] devised I/O signature identifiers, an

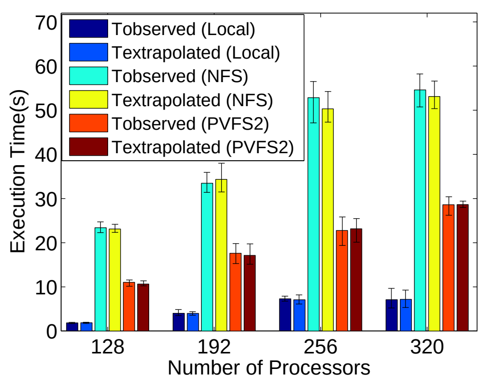

Figure 6: Results of IOR file-per-processor

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. 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 [18] 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 number of ranks.

Eckert and Nutt [3, 2] studied the extrapolation of trace data of multiple threaded programs on shared memory multiprocessors. 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 [9] assessed different trace reduction techniques. Their similarity metric (performance) resembles our wall-clock time. Their per-core metrics lack scalability. If they were enhanced so that they scaled, they would be similar to our histograms. Their compression reduction (based on flat distances) does not capture ScalaTrace's structurally recursive compression.

7. CONCLUSION

We presented the design and implementation of the extrapolation tool ScalaIOExtrap. 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.

Our results demonstrate that we preserve event ordering and time accuracy in these large traces. With this technique, large-scale I/O performance evaluation can be performed without executing the target application at scale. We can conclude that our approach opens up new opportunities for I/O performance analysis as we have the capability of extrapolating traces to arbitrary number of ranks from a set of smaller traces while retaining correct access patterns, I/O size, and I/O operations. We have demonstrated that we can also retain execution times of trace up to 320 processors for representative HPC I/O workloads.

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. The research used resources from Argonne Leadership Computing Facility (ALCF).

8. REFERENCES

[1] 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.

[2] 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.

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

[4] 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.

[5] 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.

[6] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock. I/O performance challenges at leadership scale. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 40. ACM, 2009.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] H. Shan, K. Antypas, and J. Shalf. Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark. In *Proceedings of Supercomputing*, November 2008.

[16] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective i/o in romio. In *7th Symposium on the Frontiers of Massively Parallel Computation, 1999. Frontiers '99.*, pages 182–189, Feb 1999.

[17] 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.

[18] 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.

[19] X. Wu and F. Mueller. Scalaextrap: Trace-based communication extrapolation for SPMD programs. *SIGPLAN Not.*, 46(8), Feb. 2011.

[20] 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.

[21] 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.