IOPin: Runtime Profiling of Parallel I/O in HPC Systems

Seong Jo Kim*, Seung Woo Son†, Wei-keng Liao†, Mahmut Kandemir*, Rajeev Thakur‡, and Alok Choudhary†

* Computer Science and Engineering
Pennsylvania State University
University Park, PA 16802, USA
{seokim,kandemir}@cse.psu.edu

† Electrical Engineering and Computer Science
Northwestern University
Evanston, IL 60208, USA
{sson,wkliao,choudhar}@eecs.northwestern.edu

‡ Mathematics and Computer Science
Argonne National Laboratory
Argonne, IL 60439, USA
thakur@mcs.anl.gov

***Abstract*—Many I/O- and data-intensive scientific applications use parallel I/O software to access files in high performance. On modern parallel machines, the I/O software consists of several layers, including high-level libraries such as Parallel netCDF and HDF, middleware such as MPI-IO, and low-level POSIX interface supported by the file systems. For the I/O software developers, ensuring data flow is important among these software layers with performance close to the hardware limits. This task requires understanding the design of individual libraries and the characteristics of data flow among them. In this paper, we propose a dynamic instrumentation framework that can be used to understand the complex interactions across different I/O layers from applications to the underlying parallel file systems. Our preliminary experience indicates that the costs of using the proposed dynamic instrumentation is about 7% of the application execution time.**

I. Introduction

Users of high-performance computing (HPC) systems often find that the main performance-limiting factor for applications is the storage systems, not the CPU, memory, or network. That is, I/O behavior is the primary factor that determines the overall performance of many HPC applications. Therefore, understanding complex parallel I/O operations and the involved issues is critically important to meet the requirements for a particular system or decide an I/O solution to accommodate expected workloads.

Unfortunately, understanding I/O behavior is not trivial since it is a result of complex interactions between the hardware and a number of software layers, collectively referred to as the *I/O software stack*. Figure 1 illustrates a typical I/O stack for an HPC system. At the lowest level is the storage hardware. Above the storage hardware are the parallel file systems, such as PVFS [1], GPFS [2], Lustre [3], and PanFS [4]. The roles of the parallel file system are to manage the data on the storage hardware, to present the data as a directory hierarchy, and to coordinate accesses to files and directories. The MPI-IO library [5] sits on top of the parallel file systems. It provides a standard I/O interface and a suite of optimizations such as data caching, process coordination, and so on [6], [7], [8], [9], [10], [11].

While the MPI-IO interface is effective and advantageous

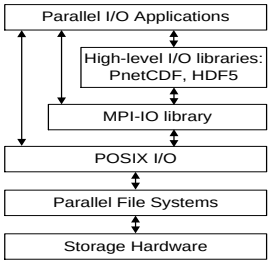


Fig. 1. Parallel I/O software stack

thanks to its performance and portability, it does not support structured data abstraction for scientific applications. To provide such structured data format, high-level I/O libraries (e.g., Parallel netCDF [12] and HDF5 [13]) are added on top of MPI-IO. As shown in Figure 1, a parallel I/O application may directly call an MPI-IO library or a POSIX I/O function to access the disk-resident data sets, as an alternative. Since the interactions among these layers are complex and unpredictable, understanding and characterizing those interactions must precede performance tuning and optimization for the HPC applications.

One approach to understanding I/O behavior is to let application programmers or scientists manually instrument the I/O software stack. Unfortunately, this approach is extremely difficult and error-prone. In fact, instrumenting even a single I/O call may necessitate modifications to numerous files to trace it from the application to multiple I/O software layers below. Worse, a high-level I/O call from the application program can be fragmented into multiple calls (sub-calls) in the MPI library, which is severely challenging. Since most parallel scientific applications today are expected to run on large-scale systems with hundreds of thousands of processors in order to achieve better resolution, even collecting and analyzing trace information from them are laborious and burdensome.

Motivated by these observations, we have developed a *dynamic* performance visualization and analysis tool for parallel I/O, called *IOPin*. Instead of manually instrumenting applications and other components of the I/O stack, we leverage a lightweight binary instrumentation using probe mode in Pin [14] to implement our current prototype. That is, our tool

performs the instrumentation with minimal overhead in the binary code of the MPI library and the underlying parallel file system at runtime. Therefore, our tool provides the language-independent instrumentation targeting scientific applications written in C/C++ and Fortran. Furthermore, our tool requires neither source code modification nor recompilation of the application and the I/O stack components.

A unique aspect of our implementation is that it provides a hierarchical view for parallel I/O. In our implementation, each MPI I/O call has a unique identification number in the MPI-IO layer and is passed to the underlying file system with trace information. This mechanism helps associate the MPI I/O call issued from the applications with its sub-calls in the PVFS layer in a systematic way. In addition, our tool provides detailed *I/O performance metrics* for each I/O call, including I/O latency at each I/O software stack layer, the number of disk accesses, disk throughput, the number of I/O calls issued to the PVFS server.

The rest of this paper is organized as follows. Related work is discussed in Section II and background for Pin framework is given in Section III. An overview of our dynamic instrumentation scheme is presented in Section IV, and the technical details and computation methodology are explained in Section V. An experimental evaluation of the tool is presented in Section VI, followed by our concluding remarks in Section VII.

II. Related Work

Over the past decade, a lot of static/dynamic code instrumentation tools have been developed and tested that target different machines and applications. Static instrumentation generally inserts some sort of probe code into the program at compile time. Dynamic instrumentation, on the other hands, intercepts the execution of binary at different points of execution and inserts instrumentation code at runtime. ATOM [15] statically instruments the binary code through rewriting at compile time. Static instrumentation is also supported in [16] to trace parallel I/O calls from the MPI library to PVFS servers. HP's Dynamo [17] monitors an executable's behavior through interpretation and dynamically selects hot instruction traces from the running program. DynamoRIO [18] is a binary package with an interface for both dynamic instrumentation and optimization. In comparison, Daikon [19] uses instrumentation to extract program invariants.

Several techniques have been proposed in the literature to reduce instrumentation overheads. Dyninst [20] and Paradyn [21], designed for dynamic instrumentation, employ fast breakpoints to reduce the overheads incurred during instrumentation. INS-OP [22] is also a dynamic instrumentation tool that applies transformations to reduce the overheads in the instrumentation code.

Tools such as CHARISMA [23], Pablo [24], and Tuning and Analysis Utilities (TAU) [25] collect and analyze file system traces [26]. Darshan [27] captures I/O behavior such as file access patterns in applications, and Vampir [28] provides an analysis framework for MPI applications. Stack Trace Analysis Tool (STAT) [29] is designed to help debug large-scale parallel programs. HPCToolkit [30] also uses sampling for measurement and analysis of program performance.

For the MPI-based parallel applications, several tools have been developed, such as MPI Parallel Environment (MPE) [31] and mpiP [32].

Our work differs from these efforts primarily because we provide a *dynamic* instrumentation framework to entirely trace parallel I/O from the MPI library to the underlying parallel file system. Our Pin-based implementation inserts instrumentation and analysis code, and profiles the parallel I/O performance at runtime. Further, we support various analytical functionalities and metrics such as latency, disk throughput, the number of disk accesses to investigate detailed I/O behavior.

III. Background

Pin is a software system that performs runtime binary instrumentation of Linux and Window applications. The goal of Pin is to provide an instrumentation platform for implementing a variety of program analysis tools for multiple architectures. In Pin, one may add analysis routines to the application process and write instrumentation routines to determine where the analysis routines are called. Pin also provides a limited ability to alter the program behavior by allowing an analysis routine to overwrite the registers and memory.

Instrumentation is performed by a just-in-time (JIT) compiler. Pin intercepts the execution of the first instruction of the executable and generates ("compiles") new code for the straight-line code sequence starting at this instruction. It then transfers control to the generated sequence. The generated code sequence is almost identical to the original one, but Pin ensures that it regains control when a branch exits the sequence. After regaining control, Pin generates more code for the branch target and continues execution. Every time JIT fetches some code, Pintool has the opportunity to instrument it before it is translated for execution. Our initial evaluation of parallel I/O applications in JIT mode shows that the incurred overhead ranges from 38.7% to 78% of the total application execution time with the process counts of 32, 64, 128, and 256.

Application binary is also instrumented in Pin probe mode. This mode employs Pin to insert probes at the start of specified routines. Here, a probe is a jump instruction that overwrites an original instruction in the application. Before the probe is inserted, the first few instructions of the specified routine are relocated. Pin copies and translates the original bytes of the application binary and then the probe redirects the flow of control to the replacement function. After instrumentation the control flow returns to the original function. Therefore, the application and the replacement routine are run natively in probe mode. This improves performance, but also puts more responsibility on the tool writer. In this work, IOPin is implemented in probe mode.

IV. Overview of Dynamic Instrumentation

The main goal behind this work is to understand the I/O characteristics of parallel applications, by detecting a "critical

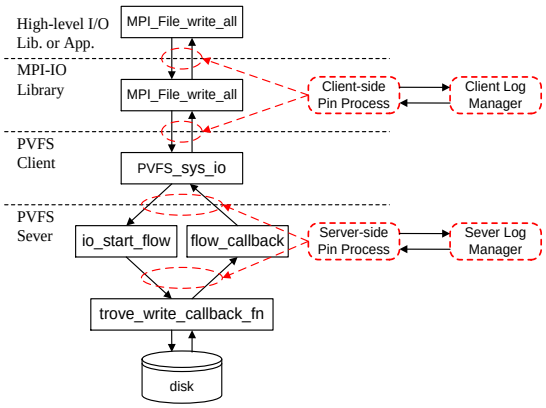


Fig. 2. Overview of our dynamic instrumentation framework. The client Pin process creates trace information for the MPI library and PVFS client at the boundary of each layer, and send it to the client log manager. The server Pin process produces trace information—the latency spent in the server, processed bytes, the number of disk accesses, and I/O throughput—and transmits it to the server log manager.

I/O path" at runtime from the process to the parallel file system that affects the entire system performance. Based on the knowledge about I/O behavior, application programmers and scientists can optimize performance by redesigning applications or system architecture. Our current prototype exploits Pin [14], a lightweight binary instrumentation tool to instrument the binary code of the MPI library and PVFS. As a result, our tool does not require source code modification and recompilation of the I/O software stack components.

Figure 2 shows the overview of our Pin-based framework. This figure is intended to explain the flow of MPI I/O call and how the framework carries out the dynamic instrumentation when a collective write function is issued. In the figure, two Pin profiling processes on the client side and the server side generate trace log information at the border of each layer—the MPI library, PVFS client, and PVFS server. The log on the client side contains trace information of each layer such as rank, mpi_call_id, pvfs_call_id, I/O type (read/write), and latency spent in the MPI library and PVFS client. In the server log with these metrics, additional information is also sent to the server log manager, such as pvfs_server_id, latency in server, bytes to be read/written, the number of disk accesses, and disk throughput for the MPI I/O call at runtime.

Both log managers are implemented in SQLite [33], a software library that implements a SQL database engine. Each log manager sends the record information back to the corresponding Pin process that has a maximum latency for the I/O operation. Then, the Pin identifies the process that has a maximum I/O latency from it, and traces and instruments only this process. This selective dynamic instrumentation not only reduces overheads, but also detects only one critical I/O path that affects the system performance effectively in the I/O stack.

V. Technical Details

We provide here details about dynamic code instrumentation and computation methodology for latency and throughput.

A. Detailed Dynamic Instrumentation

Figure 3 illustrates in detail how our implementation performs dynamic instrumentation. When an MPI I/O function call is issued from the high-level I/O library or application, the Pin process on the client side generates trace information, including rank, mpi_call_id, pvfs_call_id, I/O types (read/write), and timestamp in the MPI library. By definition, the MPI I/O function call is replaced with `PVFS_sys_io` function in the MPI library with additional arguments (`PVFS_IO_WRITE` and `PVFS_HINT_NULL`) to be issued to PVFS client. Here, the Pin process packs the trace information into a `PVFS_hints` structure and replaces the last argument, PVFS_HINT_NULL (intially set to NULL by default), with the Pin-customized hint in the PVFS_sys_io(). In the PVFS client, the Pin process extracts the trace information from `hints` and stores the trace information in the buffer to calculate latency later. The Pin-defined hint is encapsulated into a state machine control block (`smcb`) structure and passed to the PVFS server.

At the starting point of server, the Pin process searches a customized PVFS_hints from the first argument (`*smcb`) and extracts the trace information. For each I/O operation, PVFS server maintains a `flow_descriptor` structure from smcb. This flow_descriptor includes all information about the corresponding I/O request and flows until the end of the I/O operation. Since the Pin-customized hint containing the trace information exists in flow_descriptor, the server Pin process can extract it from `hints` in flow_descriptor at any point in the server without complexity.

At the entry point of disk write operation, `trove_write_callback_fn()`, the Pin process acquires the address that points to the flow_descriptor from the first argument (`void *user_ptr`) in the function. It then finds the PVFS_hints from it and stores disk I/O information, including the bytes processed, the number of disk accesses at the end of the disk operation with the corresponding rank, and the id information extracted from `hints`.

At the exit point of the server, Pin produces the log information with necessary information, e.g., rank, mpi_call_id, pvfs_call_id, I/O type, bytes processed for the corresponding MPI I/O operation, the number of accesses to disk, latency spent in the server, and disk throughput. This server log information is sent to the server log manager. Again, the Pin process on the client side generates a log at the exit point of the layer and sends it to the client log manager.

The client log manager sends the record information back to the client Pin process that has a maximum latency for the I/O operation. The client Pin detects the MPI process that has the maximum I/O latency, and traces and instruments *only* this process. The server side Pin also identifies the I/O server that spends the longest time to handle the I/O request. Our selective dynamic instrumentation not only reduces the overhead, but also effectively detects only one "critical I/O path" to the server among hundreds of thousands processes that affects the

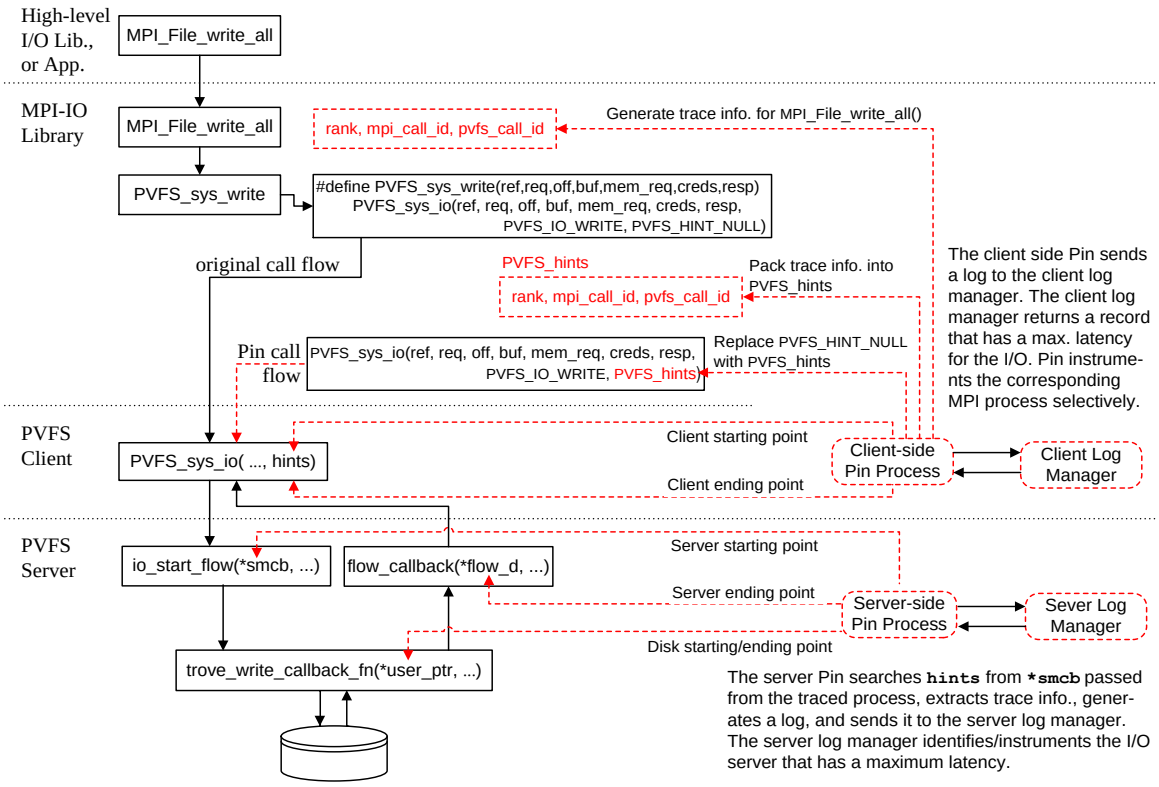


Fig. 3. Detailed illustration of how the trace information is passed. The Pin process creates a PVFS_hints structure that contains rank, mpi_call_id, and pvfs_call_id. It then replaces PVFS_HINT_NULL in PVFS_sys_io() with the Pin-generated PVFS_hints.

system performance in the I/O software stack.

At the end of the execution, by simply associating the mpi_call_id and pvfs_call_id in the client with the one in the server, the entire I/O path from the MPI library to PVFS server can be traced with the performance metrics. The detailed computation methodology for the performance metrics is explained in the next section.

B. Computation Methodology

To help users understand and analyze I/O behavior for the scientific applications, our tool provides performance statistics such as latency, disk throughput, the number of I/O requests from the client to the server, and the number of disk accesses for the I/O request. Figure 4 illustrates the computation of latency and throughput. For each I/O operation, the value of I/O latency computed at each layer is the *maximum* of the I/O latencies from the layers below it:

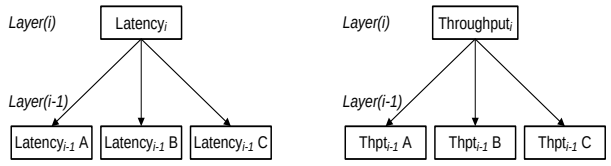
$$Latency_i = Max(Ltn_{i-1}A, Ltn_{i-1}B, Ltn_{i-1}C).$$

However, the computation of I/O throughput in Figure 4(b) is additive; in other words, the I/O throughput computed at any layer is the *sum* of the I/O throughput from the layers below it:

$$Throughput_i = \sum(Thpt_{i-1}A, Thpt_{i-1}B, Thpt_{i-1}C).$$

VI. Evaluation

Our dynamic instrumentation framework for the parallel I/O application is evaluated on the Breadboard [34] cluster at Argonne National Laboratory (ANL). Each node of this

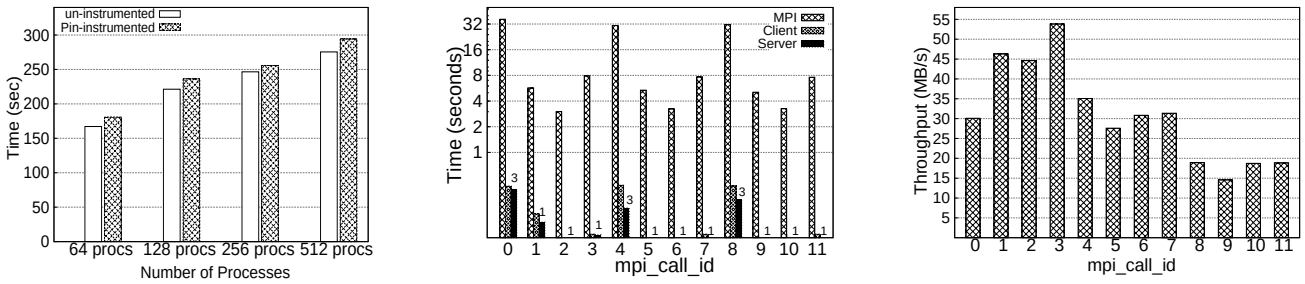


(a) Computation of I/O latency (b) Computation of I/O throughput

Fig. 4. Computation of latency and throughput. I/O latency computed at each layer is equal to the maximum value of the I/O latencies obtained from the layers below it. In contrast, I/O throughput is the sum of I/O throughput coming from the layers below.

cluster consists 8 quad-core Intel Xeon Processors and 16 GB main memory. Therefore, each physical node can support 32 MPI processes. We evaluated our implementation running on 1 metadata server, 8 I/O servers, and 256 processes. In our evaluation, we use pnetcdf-1.2.0 as a high-level I/O library, mpich2-1.4 as a middleware, and pvfs-2.8.2 as a parallel file system. To demonstrate the effectiveness of the framework, we tested a I/O-intensive benchmark, S3D-IO [35].

S3D I/O is the I/O kernel of S3D application, a parallel turbulent combustion application using a direct numerical simulation solver developed at Sandia National Laboratories (SNL). A checkpoint is performed at regular intervals; its data consists primarily of the solved variables in 8-byte, three-dimensional arrays. At each checkpoint, four global arrays—representing the variables of mass, velocity, pressure, and temperature—are written to files. All four arrays share the same size for the lowest three spatial dimensions X, Y, and



(a) Comparison of S3D I/O execution time. (b) Execution time of S3D I/O. (c) I/O throughput of S3D I/O.

Fig. 5. Comprehensive results drawn by IOPin. Figure (a) shows the execution time running on un-instrumented I/O stack and Pin-instrumented. The overhead caused by Pin-instrumentation in probe mode is about 7%, on average. In (b), for each mpi_call_id, the latency spent in MPI, client, and server is plotted in order. The latencies for some mpi_call_id's in client and server are barely seen because they are less than 0.1 sec. The figure on the server bar implies the number of fragmented calls (sub-calls). In (c), the throughput of mpi_call_id 0, 4, and 8 is plotted cumulatively as in V-B, even if they are split into 3 sub-calls.

Z and are partitioned among the MPI processes along with X-Y-Z dimensions. In our evaluation, we maintain the block size of the partitioned X-Y-Z dimension as $200 \times 200 \times 200$ in each process. With the PnetCDF interface, it produces three checkpoint files, 976.6MB each.

Figure 5(a) compares the execution time of S3D I/O when running on un-instrumented I/O stack and dynamically instrumented I/O stack. We observe that, with the process counts of 32, 64, 128, and 256, the average overhead incurred by our proposed dynamic instrumentation is about 7%.

Plotted in Figure 5(b) is the latency spent in the MPI library, PVFS client, and PVFS server from the perspective of one of the aggregator processes on a critical I/O path among 256 processes. Note that a large fraction of the time spent in the server is for disk operations even though not shown here. In S3D I/O, three checkpoint files are produced by 12 collective I/O calls, and each checkpoint file is generated by 0∼3, 4∼7, and 8∼11, respectively. For example, the first checkpoint file is opened by mpi_call_id 0. The four arrays of mass, velocity, pressure, and temperature are sequentially written by the mpi_call_id 0, 1, 2, and 3. We observe from Figure 5(b) the latency difference between the MPI library and the PVFS client. During the collective I/O operation in S3D I/O, all the joined processes heavily exchange data for optimizations such as data sieving [10] and two-phase I/O [11]. In addition, communication and synchronization among the processes cause the overhead in the MPI library. We also notice that the latency in the MPI library for mpi_call_id 0, 4, and 8 is longer than that of the others. These calls are to open the individual checkpoint file and to write the mass value which is the largest array among the four. In our experiment, these calls are fragmented into 3 sub-calls to satisfy the I/O requests. The figure on the server bar in Figure 5(b) indicates the number of fragemented calls which is also the number of disk accesses.

Figure 5(c) plots the throughput of an individual I/O call from mpi_call_id 0 to 11. The first calls (0, 4, and 8) to create the invididual checkpoint file are split into 3 sub-calls, respectively, and the throughput of those I/O calls is plotted cumulatively as explained in V-B. We observe that the I/O throughput to for creating and writing the first file is higher than the others, on average, which needs to be further investigated.

Based on the understanding of I/O characteristic from the given applications, scientists and application programmers can customize the existing application code to better use the middleware. Also, performance engineers may reduce the the overhead caused by such optimizations in the MPI library.

VII. Conclusions and Future Work

Understanding I/O behavior is one of the most important steps for efficient execution of data-intensive scientific applications. The first step in understanding I/O behavior is to instrument the flow of an I/O call. Unfortunately, performing manual instrumentation is extremely difficult and error-prone since the characteristics of I/O are a result of complex interactions of both hardware and multiple layers of software components. Because of the scale of the current HPC systems, collecting and analyzing trace information are challenging and daunting tasks. To alleviate these difficulties, we propose a dynamic instrumentation framework working on the binary code of the MPI library and PVFS. The tool inserts trace information into a PVFS_hints structure and passes it into the sub-layers at runtime. This method can provide a hierarchical view of the I/O call from the MPI library to the PVFS server without source code modification or recompilation of the I/O stack.

We used a scientific application benchmark, S3D I/O, to evaluate our proposed framework. Changing the number of processes to run S3D I/O, we made different experiments and observed that the overhead induced by our implementation is about 7% on average. Our tool provides several metrics to understand and analyze I/O behavior, such as the latency of each layer, the number of fragmented I/O calls and disk accesses, and I/O throughput. The results from these metrics contribute to evaluating and tuning the applications and I/O software stack. Work is underway (i) to test our framework under very large process counts, and (ii) to employ our framework for runtime (dynamic) I/O optimizations.

REFERENCES

[1] P. Carns, W. Ligon III, R. Ross, and R. Thakur, "PVFS: A parallel file system for Linux clusters," in *Proceedings of the 4th annual Linux Showcase & Conference-Volume 4*. USENIX Association, 2000, pp. 28–28.

[2] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proceedings of the First USENIX Conference on File and Storage Technologies*. Citeseer, 2002, pp. 231–244.

[3] P. Schwan, "Lustre: Building a file system for 1000-node clusters," in *Proceedings of the 2003 Linux Symposium*. Citeseer, 2003.

[4] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable performance of the Panasas parallel file system," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*. USENIX Association, 2008, p. 2.

[5] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir, "MPI - The Complete Reference: Volume 2, The MPI-2 Extensions," 1998.

[6] A. Ching, A. Choudhary, K. Coloma, W. Liao, R. Ross, and W. Gropp, "Noncontiguous I/O accesses through MPI-IO," 2003.

[7] X. Ma, M. Winslett, J. Lee, and S. Yu, "Improving MPI-IO output performance with active buffering plus threads," 2003.

[8] K. Coloma, A. Choudhary, W. Liao, L. Ward, E. Russell, and N. Pundit, "Scalable high-level caching for parallel I/O," 2004.

[9] W. Liao, A. Ching, K. Coloma, A. Choudhary *et al.*, "An implementation and evaluation of client-side file caching for MPI-IO," in *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2007, p. 49.

[10] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective I/O in ROMIO," in *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*, 1998, pp. 182–189.

[11] J. Del Rosario, R. Bordawekar, and A. Choudhary, "Improved parallel I/O via a two-phase run-time access strategy," *ACM SIGARCH Computer Architecture News*, vol. 21, no. 5, pp. 31–38, 1993.

[12] J. Li, W. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel netCDF: A high-performance scientific I/O interface," in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2003, p. 39.

[13] "HDF5: Hierarchical Data Format," http://www.hdfgroup.org/HDF5/.

[14] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *ACM SIGPLAN Notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[15] A. Srivastava and A. Eustace, "ATOM: A system for building customized program analysis tools," in *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*. ACM, 1994, pp. 196–205.

[16] S. Kim, Y. Zhang, S. Son, R. Prabhakar, M. Kandemir, C. Patrick, W. Liao, and A. Choudhary, "Automated tracing of i/o stack," *Recent Advances in the Message Passing Interface*, pp. 72–81, 2010.

[17] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: a transparent dynamic optimization system," in *ACM SIGPLAN Notices*, vol. 35, no. 5. ACM, 2000, pp. 1–12.

[18] D. Bruening, "Efficient, transparent, and comprehensive runtime code manipulation," Ph.D. dissertation, MIT, 2004.

[19] M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1-3, pp. 35–45, 2007.

[20] O. Source, "Dyninst: An application program interface (api) for runtime code generation," *Online, http://www. dyninst. org*.

[21] J. Hollingsworth, O. Niam, B. Miller, Z. Xu, M. Gonçalves, and L. Zheng, "MDL: a language and compiler for dynamic program instrumentation," in *PACT'97*. Published by the IEEE Computer Society, 1997, p. 201.

[22] N. Kumar, B. Childers, and M. Soffa, "Low overhead program monitoring and profiling," in *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. ACM, 2005, pp. 28–34.

[23] N. Nieuwejaar, D. Kotz, A. Purakayastha, S. Ellis, and M. Best, "File-access characteristics of parallel scientific workloads," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 10, pp. 1075–1089, 1996.

[24] H. Simitci, "Pablo MPI Instrumentation User's Guide," 1996.

[25] S. Moore, F. Wolf, J. Dongarra, S. Shende, A. Malony, and B. Mohr, "A scalable approach to MPI application performance analysis," *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pp. 309–316, 2005.

[26] S. Browne, J. Dongarra, and K. London, "Review of performance analysis tools for MPI parallel programs," *NHSE Review*, vol. 3, no. 1, pp. 241–248, 1998.

[27] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 characterization of petascale I/O workloads," in *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*. IEEE, 2009, pp. 1–10.

[28] W. Nagel, A. Arnold, M. Weber, W. Nagel, A. Arnold, M. Weber, and K. Solchenbach, "VAMPIR: Visualization and analysis of MPI resources," 1996.

[29] D. Arnold, D. Ahn, B. De Supinski, G. Lee, B. Miller, and M. Schulz, "Stack trace analysis for large scale debugging," in *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2007, p. 64.

[30] J. Mellor-Crummey, "Hpctoolkit: Multi-platform tools for profile-based performance analysis," in *5th International Workshop on Automatic Performance Analysis (APART)*. Phoenix, AZ, November, 2003.

[31] A. Chan, W. Gropp, and E. Lusk, "Users guide for MPE: extensions for MPI programs," Argonne National Laboratory, Argonne, Illinois, Tech. Rep. ANL/MCS-TM-ANL-98/xx, 2003.

[32] J. Vetter and C. Chambreau, "mpiP: Lightweight, Scalable MPI Profiling," URL: http://www.llnl.gov/CASC/mpiP.

[33] SQLite. [Online]. Available: http://www.sqlite.org/

[34] http://wiki.mcs.anl.gov/radix/index.php/Breadboard.

[35] R. Sankaran, E. Hawkes, J. Chen, T. Lu, and C. Law, "Direct numerical simulations of turbulent lean premixed combustion," in *Journal of Physics: conference series*, vol. 46. IOP Publishing, 2006, p. 38.