DXT: Darshan eXtended Tracing

Cong Xu*, Shane Snyder†, Omkar Kulkarni*, Vishwanath Venkatesan*,
Philip Carns†, Suren Byna‡, Robert Sisneros§, and Kalyana Chadalavada*

*Intel Corporation Email: {cong.xu,omkar.kulkarni,vishwanath.venkatesan,kalyana.chadalavada}@intel.com
†Argonne National Laboratory Email: {ssnyder,carns}@mcs.anl.gov
‡Lawrence Berkeley National Laboratory Email: sbyna@lbl.gov
§National Center for Supercomputing Applications Email: sisneros@illinois.edu

***Abstract*—As modern supercomputers evolve to exascale, their I/O subsystems are becoming increasingly complex, making optimization of I/O for scientific applications a daunting task. Although I/O profiling tools facilitate the process of optimizing application I/O performance, legacy profiling tools lack flexibility in their level of detail and ability to correlate traces with other sources of data. Additionally, a lack of robust trace analysis tools makes it difficult to derive actionable insights from large-scale I/O traces.**

Darshan is an HPC I/O characterization tool that records statistics in a lightweight manner that makes it appropriate for full-time production deployment. However, Darshan's default characterization mechanism records information at a fixed granularity. We augment Darshan by proposing Darshan eXtended Tracing (DXT) for more detailed profiling of I/O software stacks. DXT enables users and administrators to vary the level of fidelity captured by Darshan at run time without modifying or recompiling applications. This capability facilitates systematic analysis on the I/O behavior of applications and can provide useful application kernel I/O traces to help advance parallel I/O research. We have demonstrated the power of DXT by obtaining a wide range of useful statistics for multiple case studies, and we further show that DXT is able to do so same with negligible overhead.

I. Introduction

The past decade has witnessed an exponential increase in scientific data produced on high-performance computing (HPC) systems. Across different disciplines, scientific applications adopt a broad range of strategies to store colossal amounts of data to storage subsystems. Multiple studies have found, however, that the I/O performance of production applications often suffers considerably due to unforeseen factors in practice [14], [7]. Analyzing the cause of poor I/O performance is a complex undertaking, especially in large-scale parallel I/O systems comprised of multilayered software stacks and hardware components.

I/O profiling tools have been developed to cope with such complexities and are employed to characterize the I/O activities carried out on parallel file systems. Legacy profiling tools, however, either provide limited information with coarse granularity or introduce unacceptable overheads in terms of application runtime, memory footprint or disk space usage. Furthermore, if users or administrators wish to alter the level of profiling, they often must explicitly adopt different characterization tools with diverse usage conventions and output formats. There is a clear need for a comprehensive profiling tool that can generate both high-level statistics and detailed I/O traces with minimal overhead and allow the user to tailor the output according to specific requirements.

Darshan [9] is a user-level, scalable I/O characterization tool widely deployed on large-scale HPC systems. It intercepts I/O function calls at multiple levels within the application I/O path, collects I/O traces, and reports aggregated statistics. We have extended Darshan and propose Darshan eXtended Tracing (DXT) to provide high resolution traces of application I/O which can be used to study behaviors of a wide range of workloads. DXT is intended to be a generic I/O profiling tool catering to almost any underlying file system. It is disabled by default and may be enabled by setting an environment variable. Through multiple experiments over various I/O access patterns, it was found that the overhead introduced by DXT was less than 1%.

Using the bundled tool, DXT logs can be parsed, analyzed and visualized offline, i.e. on a separate system or node from the ones on which the job was run to minimize the impact on application performance. DXT can intercept a range of calls for the most commonly used I/O APIs, namely POSIX and MPI-IO, thereby allowing us to perform comprehensive analysis. We have implemented many features in the analysis tool to provide useful insights into applications' I/O access patterns and their performance with respect to the underlying file system. These features include noncontiguous I/O detection, data distribution analysis, I/O bandwidth reporting, and outlier detection.

In addition, since DXT can provide detailed I/O tracing information with negligible overhead, we can deploy it on any large-scale cluster without significantly affecting regular workflows of users. These traces can be used to study the application's I/O behavior from the file system's perspective by correlating it with the physical layout of the file and other tunables.

In this paper, we present the design and implementation details of the lightweight I/O profiling tool DXT. Based on the detailed I/O tracing information collected by DXT, we have developed log parsing and visualization scripts to conduct systematic analysis. Through two case studies we demonstrate that DXT can provide users with interesting insights into application I/O behaviors.

The rest of the paper is organized as follows. Section II provides the background on existing I/O characterization tools and our motivation for the DXT effort. We then describe the design of DXT components and our strategies to limit their overhead in Section III. Section IV measures the performance and scalability of DXT, followed by two case studies in Section V. In section VI we present the existing research related to our work. Finally, we conclude the paper in section VII.

II. Background and Motivation

I/O libraries such as MPI I/O [3], HDF5 [11], and netCDF [5], provide an efficient way to represent complex data structures in applications. The last decade has seen an increased interest in using such libraries amongst the scientific computing community that makes it imperative to optimize these libraries across file systems. Optimizing such middleware has been done in the past by treating the underlying file system as a black box by predicting file system behavior based on empirical analysis. LIOProf [15], tries to address this problem by providing a solution to provide feedback from the file system on the behavior of the middleware. This can be used to optimize the way I/O is sent to the file system from the middleware thereby improving the performance of the middleware. The LIOprof approach, although useful, is restricted to Lustre [1] and also requires super user privileges to obtain such traces.

The Darshan eXtended Tracing (DXT) extends Darshan [9] which is an I/O characterization tool widely adopted and used in many leadership scale systems. DXT allows us to extract detailed I/O traces from the user space in a file system agnostic way. DXT has been designed in such a way that it remains switched off by default and can be switched on by toggling an environment variable at runtime. Parsing and visualization of data on DXT traces can be done offline and such analysis can be used to identify issues like lock contention, stripe misalignment, imbalance in data distribution and detection of outliers with post-analysis tools.

One other important advantage of implementing extended tracing as a part of Darshan is that it supports intercepting I/O calls from multiple layers of the stack. This allows developers to correlate and identify issues in I/O issued in the different layers. For example, MPI I/O may use POSIX I/O to access the file system. DXT can provide traces for multiple layers that can be very useful in identifying potential issues in I/O patterns at a specific layer.

Despite the size of the traces, the overhead of saving the trace information is kept as low as 1% using collective buffering [10] to write traces to the parallel file system. With such minimal overhead, DXT could potentially be used to extract detailed I/O traces from applications running in leadership scale clusters. And since these traces are file system agnostic, they can be used for benchmarking and I/O research. Variety of research areas like prefetching, cache replacement policies can benefit from such traces. Such traces could also reduce the time spent on running application benchmarks and improve distribution and reproducibility of workloads. The Storage Networking Industry Association (SNIA) [8] provides these traces for aiding storage research, but the repositories lack HPC workloads and applications. Using DXT, such traces can easily be generated and added to such repositories.

III. Design and Implementation

DXT aims to provide detailed I/O tracing information of applications for users to conduct comprehensive analysis. It records all the I/O requests and related information in the MPI-IO and POSIX layers. For each I/O request, it reports the file offset, length, start and end times, as well as the issuing MPI process rank and hostname of the compute node. Like Darshan, DXT intercepts I/O function calls issued in multiple layers and does not require any modifications to the source code of applications. DXT instruments applications via either compile-time wrappers for statically linked executables or dynamic library preloading for dynamic executables.

DXT contains two main components: DXT logging and DXT analysis tool. The DXT logging component is responsible for recording the I/O activities of applications. During application execution, it temporarily stores the I/O traces in a memory buffer within each process. At the end of the job, it compresses and flushes the traces to persistent storage where they can be analyzed offline. The DXT analysis tool can be used to analyze and visualize the I/O traces collected by the DXT logging component. We have developed a command line interface to analyze and visualize useful information about factors that influence I/O performance such as the I/O bandwidth, performance outliers, data distribution on the parallel file system, noncontiguous I/O, and stripe misalignment.

As previously outlined, it is important that profiling overheads imposed by DXT's logging component be minimal to limit possible impact on applications. The most critical overheads from the application's perspective are the runtime costs of intercepting and extracting trace data for each I/O operation and allocating additional memory to store these traces. To minimize impact on the runtime, DXT avoids communication and I/O within intercepted functions, deferring these costly activities towards the end during application shutdown. For reducing memory overhead, DXT first allocates a small buffer for storing a process's trace data (around 2 KB), then gradually expanding it as needed. A user-configurable maximum buffer size (which defaults to 4 MB) is used to cap DXT's memory usage. To further reduce the memory footprint, DXT does not capture any redundant data that may otherwise be retrieved from other Darshan modules. Finally, compression is applied to reduce the size of the final output file.

The DXT analysis tool is designed to analyze and visualize I/O traces collected by DXT logging component. It can be run on any machine for offline analysis. DXT can detect the type of the file system, and accordingly query the physical file layout such as striping information. By correlating this information with DXT I/O traces, the tool is able to perform comprehensive analysis on the I/O activities of applications. For instance, the I/O traces can be grouped based on compute nodes, process ranks, Lustre OSTs, etc. which can yield interesting insights. Using statistical methods, it can report performance outliers: processes, nodes or I/O servers that may be performing sluggishly compared to the rest. The type of analysis and specific input parameters may be passed to the tool through the command line. There is also an option to output all analysis results.

IV. Evaluation

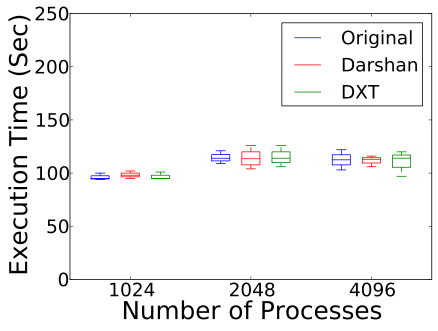
We begin our evaluation of DXT by quantifying the amount of runtime overhead that it introduces in instrumented applications. Multiple experiments were conducted on the Cori system at National Energy Research Scientific Computing Center (NERSC) and Blue Water system at the National Center for Supercomputing Applications (NCSA).

The Cori system contains two different kinds of nodes: 2,004 Intel Xeon "Haswell" nodes and 9,300 Intel Xeon Phi "Knight's Landing" nodes. The storage is a 30 PB Lustre file system with 248 Lustre OSTs. We launch our jobs on "Haswell" processor nodes. One node equips 32 CPU cores and 128 GB of memory. The Blue Waters system is a Cray XE6-XK7 supercomputing system that has 26 PB online disk capacity. There are 22,640 XE6 nodes and 4,224 XK7 nodes connected via Cray Gemini interconnect. Our tests were run on the XE6 nodes, each node has two 16 core CPUs and 64GB of main memory.

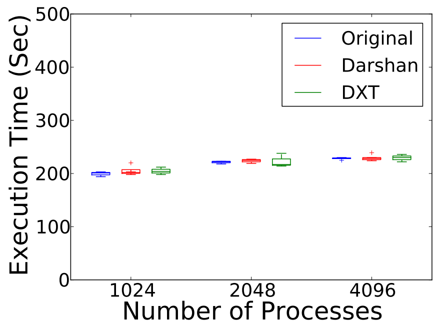
We used the synthetic workloads from the IOR benchmark for evaluations. IOR can simulate different I/O access patterns and supports multiple I/O interfaces, including POSIX and MPI-IO. It is widely used to quantify I/O performance on parallel file systems. For this experiment, we used Cray MPI (version 17.0.1) and Lustre (version 2.7.1).

A. *Overhead Measurement*

IOR was configured to use MPI-IO for quantifying the overhead introduced by DXT. Up to 4,096 processes were launched on 128 Lustre clients, interacting concurrently with 128 Lustre Object Storage Targets (OSTs). The total data size was 4 TB, and the transfer size was set to 512 KB. A directory was created on Lustre to store the DXT logs and was configured such that DXT could flush the logs to multiple Lustre OSTs.



(a) Cori System



(b) Blue Waters System

Figure 1. Overhead Measurements on Two HPC Systems

We compared the execution time of the original IOR (Original), IOR with Darshan enabled (Darshan), and IOR with DXT enabled (DXT) on both Cori and Blue Waters systems. The execution time is the time difference between "Run finished" and "Run began" reported in the IOR output. IOR was instructed to run three iterations. Fig. 1 presents the results. From the figure we can see that both the Darshan and DXT cases perform similarly to the original case at various processes on both systems. This demonstrates that both Darshan and DXT introduce negligible overhead to the run-time system. In the rest of this section, we focus on the results in Cori system.

Transfer size determines the amount of data on storage covered in each I/O function call. It is another important parameter that needs to be considered when measuring the DXT overhead. When the file size is kept constant, a smaller transfer size results in a larger number of I/O operations, consequently increasing the size of the DXT logs.

To investigate the transfer size effects on DXT overhead, we kept other parameters constant and varied the transfer size from 64 KB to 4 MB at 4,096 processes. As depicted in Fig. 2, all cases perform worst with 64 KB transfer size because it introduces too many small I/O requests to the file system. We cannot observe any differences between the

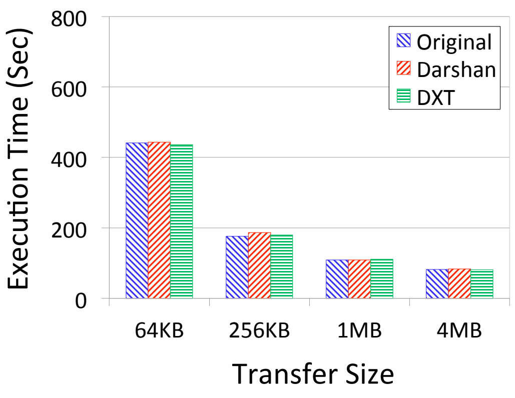


Figure 2. IOR Benchmark with Various Transfer Sizes

three cases with various transfer size from the figure because the overhead introduced by DXT is negligible.

DXT buffers the traces in the memory during the run and writes the DXT logs to Lustre at the end of the job. Therefore, any overhead (if noticeable) will be at shutdown time, since that is when the compression and output happens. In the next section we analyze the time spent in shutdown.

B. Shutdown Time

Once the application has finished executing, DXT intercepts the MPI_Finalize() function to compress and write DXT logs to Lustre concurrently. Therefore, it becomes important to investigate the impact on shutdown time since most of the overhead (if noticeable) will be incurred at this stage. Darshan can be made to report precise per-module timing metrics for flushing the logs by setting the DARSHAN_INTERNAL_TIMING environment variable.

Table I
SHUTDOWN TIME DISSECTION AT 4,096 PROCESSES

Operation	**Darshan(ms)**	**DXT(ms)**
log_open	8.22	16.39
job_write	0.47	0.34
hash_write	20.71	20.82
header_write	0.13	0.11
POSIX_shutdown	1.54	1.42
MPI-IO_shutdown	1.73	1.31
LUSTRE_shutdown	1.39	1.19
STDIO_shutdown	18.27	18.54
X_POSIX_shutdown		24.56
X_MPIIO_shutdown		99.61
Total Shutdown Time	58.48	213.27

We compared the overhead introduced by Darshan and DXT with a 512 KB transfer size at 4,096 processes. Table I presents the time spent on each operation in detail. During the shutdown time both Darshan and DXT write a log file to the parallel file system. The table shows the cost of opening the log, writing metadata information, and appending traces of each module.

Compared with Darshan, DXT introduces additional X_POSIX_shutdown and X_MPIIO_shutdown modules to provide detailed I/O tracing information in the POSIX and MPI-IO layers. As shown in the table, DXT spent 24.56ms and 99.61ms in writing the X_POSIX and X_MPIIO modules, respectively. The X_POSIX module uses less time than the X_MPIIO module because its log size is much smaller than that of the X_MPIIO module. IOR enabled a collective buffering algorithm [10] in the evaluation; the algorithm aggregated the I/O requests in the MPI-IO layer and thus issued fewer requests in the POSIX layer.

Table II
DXT SHUTDOWN OVERHEAD WITH DIFFERENT TRANSFER SIZES

Operation & Log Size	**64KB**	**256KB**	**1MB**	**4MB**
X_POSIX_shutdown (ms)	26.07	25.81	25.21	23.91
X_MPIIO_shutdown (ms)	330.80	180.93	99.42	97.04
Total Shutdown (ms)	507.21	309.45	203.66	142.33
Log Size (mb)	492.93	156.20	46.84	19.81

Table II shows the DXT shutdown time and log size with different IOR transfer sizes. As can be seen, the shutdown times of the X_POSIX module with different transfer sizes are almost the same, since the number of POSIX I/O calls remains nearly constant throughout due to aggregation of requests in collective I/O. On the other hand, for the given file size of 4 TB, the number of MPI I/O requests is inversely proportional to the transfer size resulting in larger logs for the X_MPIIO module. Consequently, both the total shutdown time and log size grow as the transfer size decreases. However, the overhead is still small; even when the transfer size is 64 KB, the 507.21 ms total shutdown cost is 0.1% of the 436 s job execution time, and the 492.93 MB DXT log size is 0.012% of the 4 TB file size.

C. Memory Usage and Log Size

The two preceding sections discussed the overall and shutdown overhead introduced by DXT. Here we focus mainly on the memory usage of DXT and the size of the log generated by DXT.

The DXT log contains both the trace header and I/O tracing segment. The trace header stores the basic information related to current process, including rank number, hostname, and number of write/read operations. The trace header is followed by the I/O tracing segment, which records the start time, end time, offset, and length of the I/O request. The memory used by each process in one I/O layer can be calculated using the following formula:

$$memory = header + io\_operations * trace\_segment \quad (1)$$

Currently the sizes of the header and trace segment are 104 Bytes and 32 Bytes, respectively. The number of I/O operations depends mainly on the number of I/O requests and the MPI-IO algorithm. In the DXT X_MPIIO module, the number of I/O operations equals the number of MPI-IO function calls issued by the application. In contrast, the number of POSIX I/O operations recorded by the DXT X_POSIX module depends on the specific MPI collective I/O algorithm. If collective buffering is enabled, the MPI-IO library aggregates I/O requests and issues actual I/O calls to the file system. Otherwise, MPI I/O calls go directly to the file system.

For instance, let us assume that IOR launches 4,096 processes to write a 4 TB file and that the transfer size is configured to be 1 MB. Then each process needs to perform 1,024 MPI-IO function calls. Thus the size of the DXT X_MPIIO module per process is 104 Bytes + 1024 * 32 Bytes, which is 32,872 Bytes. If the Lustre stripe size equals 4 MB and the collective buffering algorithm has been enabled, there will be 256 POSIX I/O function calls to the Lustre file system. The size of the POSIX I/O request is 4 MB (equal to the stripe size). Thus the size of the DXT X_POSIX module per process is 8,296 Bytes. Consequently, the total memory used by the DXT X_MPIIO and X_POSIX modules in each process equals 41,168 Bytes, which is much smaller than the 4 MB DXT memory limit per process.

Additionally, DXT employs zlib to compress the DXT logs within each process before writing to the file system. In this case we have 4,096 processes, and the total size of the DXT log is 41,168 Bytes * 4096 processes, which is 160.8 MB. After zlib compression the log size becomes 46.84 MB with a compression ratio of 3.4:1. This calculation demonstrates that the DXT log is compact to be able to buffer such a large amount of I/O tracing information in the memory.

V. Case Studies

To demonstrate the power of DXT, we use two case studies to show how DXT can provide insight into the I/O behaviors of applications. Both case studies are conducted on the Wolf development cluster at Intel Corporation. The Wolf cluster features 70 physical compute nodes, each with Octadeca-Core 2.3 GHz Xeon processors (36 cores) and 64 GB of memory. All compute nodes are connected by using Mellanox QDR ConnectX InfiniBand.

A. *GCRM-IO*

The first case study was conducted using GCRM-IO. It simulates I/O for GCRM, a global atmospheric circulation model, simulating the circulations associated with large convective clouds. The I/O kernel also uses H5Part to perform all the GCRM I/O operations with random data. The I/O pattern of GCRM-IO corresponds to a semi-structured geodesic mesh, where the grid resolution and subdomain resolution are specified as input.

To investigate the I/O performance of GCRM-IO, we launched 256 processes that wrote in parallel to a shared file on a Lustre file system. The grid and subdomain resolutions were set to 10 and 4 respectively. The total number of timesteps was 64, so the application outputs the pressure variable value 64 times. Lustre optimization and stripe alignment options provided by the application were enabled.

Lustre file system on the Wolf cluster is able to provide an aggregate write bandwidth of up to 2500 MB/s. In this case study, however, GCRM-IO was able to achieve only 1689.13 MB/s write bandwidth. We used the DXT analysis tool to study DXT logs of the application to identify the cause of the gap in performance.

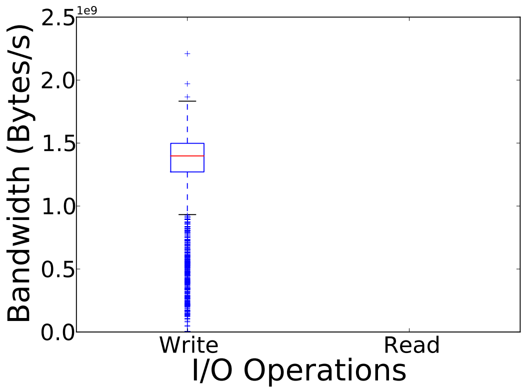
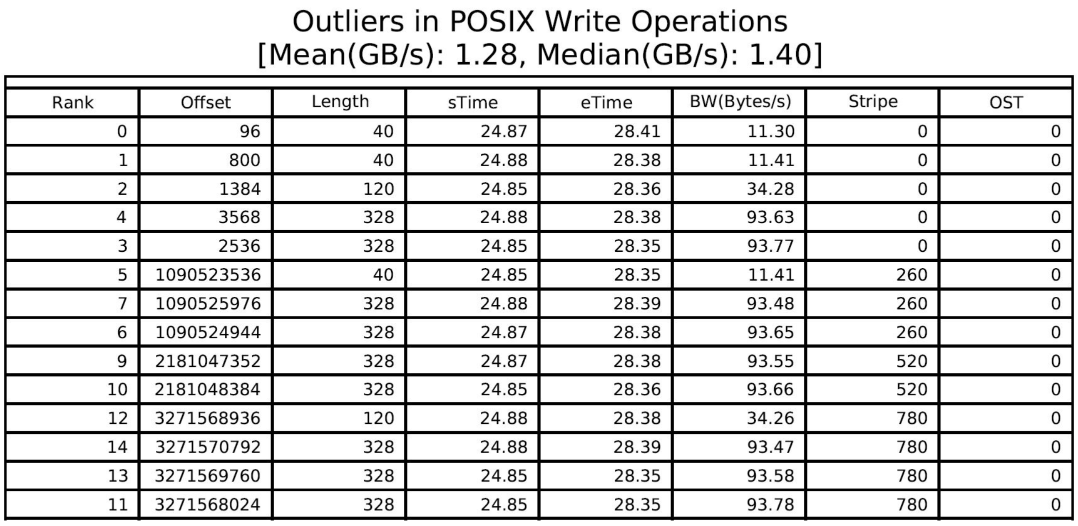


Figure 3. GCRM-IO I/O Requests Bandwidth

Fig. 3 presents a box plot of POSIX I/O requests bandwidth in GCRM-IO. The box plot whiskers extend to the far end of the x-axis (where X=1.5x of the box extents). Any results outside those extents are considered outliers. The read bandwidth is empty because GCRM-IO kernel does not read data in the evaluation. As can be observed in the figure, the median bandwidth of GCRM-IO write requests is 1.39 GB/s, and there exist plenty of outliers with inferior I/O performance.

To conduct further investigation, the DXT log analysis tool outputs the details of these outliers, Fig. 4 depicts the first part of the outputs. As shown in the figure, between 24.8 s and 28.3 s almost all processes take about 3.5 s to write a very small data segment, resulting in bad I/O performance. The major reason is that the same Lustre stripe is accessed by multiple processes simultaneously, leading to severe lock contention due to false sharing. Additionally, a number of small I/O requests issued almost concurrently to a single OST also incurs a high amount of I/O latency.

When we examine the GCRM-IO source code, we find that GCRM-IO uses the H5Part API to write HDF5 datasets collectively. However, small portions of HDF5 metadata


Outliers in POSIX Write Operations
[Mean(GB/s): 1.28, Median(GB/s): 1.40]

Rank	Offset	Length	sTime	eTime	BW(Bytes/s)	Stripe	OST
0	96	40	24.87	28.41	11.30	0	0
1	800	40	24.88	28.38	11.41	0	0
2	1384	120	24.85	28.36	34.28	0	0
4	3568	328	24.88	28.38	93.63	0	0
3	2536	328	24.85	28.35	93.77	0	0
5	1090523536	40	24.85	28.35	11.41	260	0
7	1090525976	328	24.88	28.39	93.48	260	0
6	1090524944	328	24.87	28.38	93.65	260	0
9	2181047352	328	24.87	28.38	93.55	520	0
10	2181048384	328	24.85	28.36	93.66	520	0
12	3271568936	120	24.88	28.38	34.26	780	0
14	3271570792	328	24.88	28.39	93.47	780	0
13	3271569760	328	24.85	28.35	93.58	780	0
11	3271568024	328	24.85	28.35	93.78	780	0

Figure 4. GCRM-IO I/O Requests Bandwidth Outliers

are written by each process individually, leading to huge performance penalties. To address this issue, the collective metadata I/O feature in HDF5 needs to be enabled explicitly in GCRM-IO, so that the small metadata I/O requests can be aggregated and written to the storage.

B. *HACC-IO*

The HACC-IO benchmark is the I/O kernel extracted from the Hardware Accelerated Cosmology Code (HACC) simulation. The HACC framework uses N-body techniques to simulate the formation of structure in collisionless fluids under the influence of gravity in an expanding universe.

In this case study, we launch 256 processes to perform concurrent I/O on a single shared file, the total number of particles is 8,589,934,592 which yields a file size of 304 GB. We have measured the I/O performance of two cases: POSIX I/O (POSIX) case and collective I/O (Collective) case. In POSIX I/O case, each process uses POSIX API to write a large contiguous data block directly. On the other hand, the collective I/O case employs the collective buffering algorithm to perform I/O operations.

Fig. 5 shows the I/O bandwidths of two cases. The POSIX I/O case outperforms the collective I/O case in both write and read operations. This indicates that the collective buffering algorithm is not suitable for the large contiguous I/O accesses in HACC-IO benchmarks.

To investigate the reason, we use DXT analysis tool to estimate the time spent in collective buffering algorithm, which consists of communication and I/O phases. In the communication phase, the algorithm transfers data between aggregators and all the processes. While in the I/O phase, the aggregators issue parallel I/O to the file system.

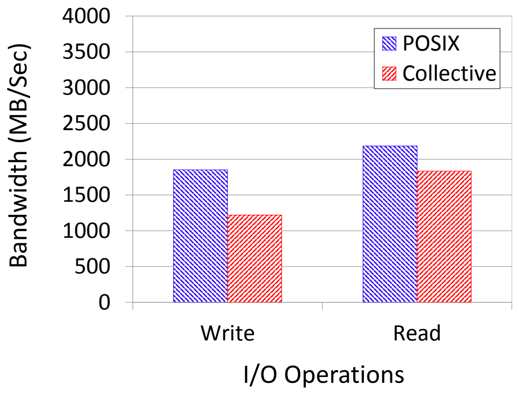


Figure 5. POSIX and Collective I/O Bandwidth Comparison

Fig. 6 presents the execution time spent in both cases. The POSIX case does not have communication cost because it performs I/O directly. In collective I/O case, the figure depicts the average communication and I/O time spent on aggregators. As shown in the figure, the I/O time has been reduced in the collective I/O case due to the effects of stripe alignment and enhanced data locality on aggregators.

However, the collective I/O case spends a large amount of time in transferring data between processes that stems from a lack of parallelism in data transfer over the network. For large contiguous requests, the data accessed by each process is striped across multiple OSTs, which are accessed through individual aggregators. At any given instant, an aggregator is only able to service an IO request from a single process while the remaining processes have to wait for their turn, leading to a serialization of network transfers. This issue is

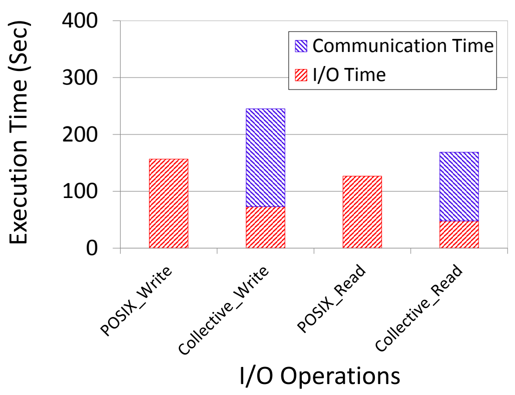


Figure 6. Time Dissection

known as aliasing and arises when the file access regions of processes are perfectly aligned such that all processes try to simultaneously access data from the same OST, and as a result, the same aggregator.

VI. Related Work

A few tools, including ScalaIOTrace [13], Recorder [6], and Darshan [9], can capture I/O tracing information in each layer of a multi-layered I/O stack. However, these tools either generate large trace files and introduce high run-time overhead or provide limited information on high-level accumulated statistics. A flexible profiling tool is needed that can generate both, detailed I/O tracing information as well as high-level statistics at the user's request.

IOPin [4] is a dynamic instrumentation framework that characterizes the I/O activities in MPI and PVFS [2]. It can provide users a hierarchical view of the I/O calls from the MPI library to the PVFS server. However, IOPin is specifically designed to profile the applications over the PVFS file system and is reported to introduce, on average, 7% overhead to the run-time systems. Clearly needed is a generic tool that can profile I/O workloads over all kinds of file systems, while introducing only minimal overhead.

The Lustre Monitoring Tool (LMT) [12] and Lustre I/O profiler (LIOProf) [15] are two important tools that provide server-side I/O profiling. LMT is responsible for monitoring the real-time status of the Lustre storage server. It serves as a Cerebro plugin on Lustre servers to collect and store server status for further analysis. LIOProf is designed to provide detailed file system I/O activities to users, helping them investigate the root cause of performance problems. Both tools require superuser (root) privileges to conduct server-side I/O profiling, and are not available to normal users.

To overcome constraints in legacy I/O profiling tools, we have developed a flexible user-level instrumentation framework DXT. It is a generic tool that is independent of the underlying file systems. It records the detailed I/O tracing information in multiple layers of I/O stacks and at the same time introduces as minimal of an overhead as possible.

VII. Conclusion

Analyzing scientific applications I/O behavior and pinpointing I/O performance bottlenecks are grand challenges for users due to increasing scale and complexity of parallel I/O systems. While a few I/O profiling tools exist to characterize the I/O activities and facilitate the analysis of I/O performance issues, a light-weight flexible instrumentation framework will serve as a powerful tool in the arsenal of scientists, application developers, system administrators and performance engineers alike.

In this paper, we have extended Darshan and proposed Darshan eXtended Tracing (DXT) to facilitate detailed I/O instrumentation on I/O software stacks. We use I/O intensive workloads to conduct systematic analysis on the overhead introduced by DXT, and demonstrate that DXT is a scalable I/O profiling tool that introduces negligible overhead to the application runtime. Through two case studies we show that DXT is capable of providing detailed I/O tracing information for comprehensive analysis of the application's I/O activities.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computer Research, under contract DE-AC02-06CH11357. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] Lustre 2.0 Operations Manual. http://wiki.lustre.org/images/3/35/821-2076-10.pdf.

[2] P. H. Carns, W. B. Ligon, III, R. B. Ross, and R. Thakur. PVFS: A Parallel File System for Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase & Conference - Volume 4*, ALS'00, pages 28–28, Berkeley, CA, USA, 2000. USENIX Association.

[3] M. P. Forum. MPI: A Message-Passing Interface Standard. Technical report, Knoxville, TN, USA, 1994.

[4] S. J. Kim, S. W. Son, W. keng Liao, M. T. Kandemir, R. Thakur, and A. N. Choudhary. IOPin: Runtime Profiling of Parallel I/O in HPC Systems. In *SC Companion*, pages 18–23. IEEE Computer Society, 2012.

[5] J. Li, W.-k. 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*, SC '03, pages 39–, New York, NY, USA, 2003. ACM.

[6] H. Luu, B. Behzad, R. Aydt, and M. Winslett. A Multi-Level Approach for Understanding I/O Activity in HPC Applications. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–5, Sept 2013.

[7] H. Luu, M. Winslett, W. Gropp, R. Ross, P. Carns, K. Harms, M. Prabhat, S. Byna, and Y. Yao. A Multiplatform Study of I/O Behavior on Petascale Supercomputers. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 33–44. ACM, 2015.

[8] SNIA - Storage Networking Industry Association. SNIA I/O Trace Data Files. http://iotta.snia.org/traces, 2017.

[9] S. Snyder, P. H. Carns, K. Harms, R. Ross, G. K. Lockwood, and N. Wright. Modular HPC I/O Characterization with Darshan. In *ESPT'16 Proceedings of the 5th Workshop on Extreme-Scale Programming Tools*, Salt Lake City, Utah, Nov. 2016. IEEE Press.

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

[11] The HDF Group. Hierarchical Data Format Version 5. https://www.hdfgroup.org/HDF5/, 2016.

[12] A. Uselton. Deploying Server-side File System Monitoring at NERSC. In *Cray User Group Conference*, 2009.

[13] 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, pages 26–31, New York, NY, USA, 2009. ACM.

[14] V. Vishwanath, M. Hereld, V. Morozov, and M. E. Papka. Topology-aware Data Movement and Staging for I/O Acceleration on Blue Gene/P Supercomputing Systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 19:1–19:11, New York, NY, USA, 2011. ACM.

[15] C. Xu, S. Byna, V. Venkatesan, R. Sisneros, O. Kulkarni, M. Chaarawi, and K. Chadalavada. LIOProf: Exposing Lustre File System Behavior for I/O Middleware. In *Cray User Group Conference*, 2016.