Modular HPC I/O Characterization with Darshan

Shane Snyder, Philip Carns, Kevin Harms, Robert Ross
Argonne National Laboratory
Lemont, IL 60439
{ssnyder,carns,rross}@mcs.anl.gov, harms@alcf.anl.gov

Glenn K. Lockwood, Nicholas J. Wright
Lawrence Berkeley National Laboratory
Berkeley, CA 94720
{glock,njwright}@lbl.gov

***Abstract*—Contemporary high-performance computing (HPC) applications encompass a broad range of distinct I/O strategies and are often executed on a number of different compute platforms in their lifetime. These large-scale HPC platforms employ increasingly complex I/O subsystems to provide a suitable level of I/O performance to applications. Tuning I/O workloads for such a system is nontrivial, and the results generally are not portable to other HPC systems. I/O profiling tools can help to address this challenge, but most existing tools only instrument specific components within the I/O subsystem that provide a limited perspective on I/O performance. The increasing diversity of scientific applications and computing platforms calls for greater flexibility and scope in I/O characterization.**

In this work, we consider how the I/O profiling tool Darshan can be improved to allow for more flexible, comprehensive instrumentation of current and future HPC I/O workloads. We evaluate the performance and scalability of our design to ensure that it is lightweight enough for full-time deployment on production HPC systems. We also present two case studies illustrating how a more comprehensive instrumentation of application I/O workloads can enable insights into I/O behavior that were not previously possible. Our results indicate that Darshan's modular instrumentation methods can provide valuable feedback to both users and system administrators, while imposing negligible overheads on user applications.

I. Introduction

The high-performance computing (HPC) community encompasses a diverse set of computational science applications, ranging from large-scale simulations of biomolecular systems to detailed models of complex weather phenomena. These applications leverage a breadth of distinct I/O strategies and I/O interfaces [1]–[3], usually according to recognized best practices in their respective problem domains. Additionally, modern HPC platforms represent numerous system architectures and storage system designs [4]–[6], each with various features and performance capabilities. The proliferation of these distinct I/O strategies and system architectures is leading to increasingly complex I/O subsystems. Unfortunately, the increasing complexity of these systems makes it difficult for application users and system administrators to reason about I/O workload performance.

I/O profiling tools can help to address this complexity by characterizing how application workloads interact with the underlying I/O subsystem. Data provided by these profiling tools can be utilized by application users to help steer potential performance-tuning efforts, while system administrators can use resultant data to evaluate systemwide I/O trends. These I/O profiling tools should, ideally, capture sufficient information to characterize I/O workloads across all layers of the stack, since I/O performance is generally dictated by the efficiency of interactions across layers rather than being isolated within specific layers. However, existing I/O profiling tools tend to focus on instrumenting specific components within the I/O subsystem and are not easily extensible to instrumenting new components. Supporting separate tools for each I/O interface or platform is intractable; thus, profiling tools need to offer flexible designs that can be easily extended to account for an ever-expanding HPC I/O stack.

In this paper we investigate how Darshan [7], an I/O profiling tool commonly enabled on some of the largest HPC systems in the world, can be modified to provide flexible and comprehensive instrumentation of the HPC I/O stack. The contributions of this work include the following:

- Modularization of Darshan's codebase to allow for instrumentation of arbitrary components in the I/O stack
- Definition of new Darshan instrumentation modules for capturing relevant I/O data from specific components in the stack
- Case studies showing how modular Darshan data can offer insight into the behavior of relevant I/O workloads
- Implementation of a novel logging mechanism allowing Darshan to generate logs for applications which terminate unexpectedly

The remainder of this paper is organized as follows. Section II provides background information on Darshan and motivates the need to modify the codebase. In Section III we provide a general overview of enhancements made to Darshan, including details on its new modularized architecture. Section IV evaluates the performance and scalability of Darshan's new design to ensure that it is still amenable to deployment on production systems. In Section V we consider case studies where Darshan enables new insights into the behavior of representative HPC I/O workloads. In Section VI we present research related to this work. Section VII provides our conclusions and outlines potential future work.

II. Background

Darshan is an application-level I/O characterization tool providing detailed statistics on the behavior of HPC I/O workloads. Rather than logging every I/O operation submitted by an application (as a tracing tool would), Darshan captures a bounded amount of data for each file opened by the application, including I/O operation counts, common I/O

978-1-5090-3918-0/16 $31.00 © 2016 IEEE
DOI 10.1109/ESPT.2016.9

access sizes, cumulative timers, and other statistical data. The total number of instrumented files is also bounded to limit Darshan's memory footprint. Prior Darshan versions[1] have instrumented in-depth data from the POSIX and MPI-IO layers of the I/O stack and basic data for the high-level HDF5 and Parallel netCDF data interface layers. Darshan also captures a fixed set of job-level parameters, such as the number of application processes, the job's start and end time, and the job ID assigned by the scheduler. Darshan can instrument I/O functions in both statically and dynamically linked executables. Wrappers are interposed at compile time using both the PMPI interface and the GNU linker's `--wrap` argument for static executables, while dynamic executables use the `LD_PRELOAD` mechanism to interpose wrappers.

Darshan's strategy for logging instrumented data to file is to defer all required communication and I/O operations until application termination. When the application terminates, Darshan performs any necessary data aggregation steps (e.g., Darshan reduces data records shared globally across all processes into a single shared record by default), then compresses each process's data before collectively writing it out to a single log file. The decision to defer communication and I/O until job shutdown, coupled with the decision to bound total memory consumption, is fundamental to Darshan's lightweight design philosophy, making it amenable to full-time deployment on production HPC systems. In fact, Darshan has been enabled by default on a number of production systems for some time, including systems at the Argonne Leadership Computing Facility (ALCF), the National Energy Research Scientific Computing Center (NERSC), and the National Center for Supercomputing Applications (NCSA). Darshan integrates directly into the build environment at these facilities to allow for transparent instrumentation of application codes. Prior research has shown how data captured by Darshan can help drive application I/O performance tuning on production systems at these facilities [7]–[9]. Darshan has also been used to enable systemwide and cross-system characterization of HPC I/O workloads [8], [9] and to generate realistic I/O workloads for evaluating the performance of storage system designs [10].

The widespread use of Darshan has also revealed several aspects of its original design that can result in incomplete *coverage*.[2] For example, the analysis by Luu et al. [9] found that Darshan was not able to provide meaningful insight into applications that perform extensive I/O using interfaces not traditionally used in HPC (e.g., via `fprintf()` provided by POSIX stdio). In addition, routine analysis of application link-time data collected by ALTD [11] at NERSC has shown that over 30% of jobs (45% of core-hours) that use Darshan do not generate a log file as a result of the application being killed because of insufficient resources (e.g., wall time) or failing with an unexpected error. The need to address these issues and extend Darshan's coverage in environments that use nontraditional I/O interfaces or enforce resource constraints has motivated new enhancements to the Darshan codebase.

[1]When referring to prior versions of Darshan, we mean versions prior to the rewrite of the codebase described in this paper (i.e., in versions prior to Darshan 3.0.0).

[2]Here, we define coverage as a broad metric representing the fraction of relevant I/O characterization data captured by Darshan from the jobs running on a specific system. Coverage is reduced when applications disable Darshan, applications fail to log Darshan data successfully or when applications use I/O interfaces not instrumented by Darshan.

III. Darshan enhancements

To address the concerns introduced in the preceding section, we identified a set of enhancements to Darshan to more flexibly and comprehensively characterize HPC application I/O workloads:

- Modularization of the Darshan runtime architecture and log file format to allow integration of data from new I/O components
- Implementation of new instrumentation modules gathering I/O workload data at different layers of the I/O stack
- A new `mmap`-based logging mechanism to improve the robustness of Darshan's data capture methods

At a high level, the primary aim of these enhancements is not only to improve Darshan's flexibility in gathering I/O characterization data but also to address existing gaps in Darshan's coverage.

A. Modularization

Darshan's original design focused on gathering data from a static set of components in the I/O subsystem, with most extracted data coming from instrumenting POSIX and MPI-IO functions of interest. However, this design did not provide mechanisms for easily instrumenting I/O data from new components, either by capturing more extensive data from the POSIX and MPI-IO interfaces or by instrumenting entirely new sources of data. For this reason, we sought to modularize Darshan's codebase by specifying new Darshan components called *instrumentation modules* that capture I/O data from a particular source (e.g., an I/O interface or file system specific API). The *Darshan core* library then exposes a well-defined interface for these instrumentation modules to coordinate with Darshan at runtime.

Instrumentation modules are responsible for the following:

- Capturing data from specific I/O components, usually by instrumenting functions of interest
- Generating I/O data records characterizing I/O behavior and registering these records with Darshan core
- Coordinating with Darshan core at shutdown to organize module records before they are written to file

On the other hand, the Darshan core library is responsible for the following:

- Initializing Darshan core data structures and allocating memory for storing module records at application start
- Providing an interface to modules for reserving memory to store records and for registering records that should be persisted in the final output log
- Mapping module record names to consistent identifiers that can be used to correlate records across modules
- Gathering final output data from modules, compressing this data and writing it to log file at application shutdown

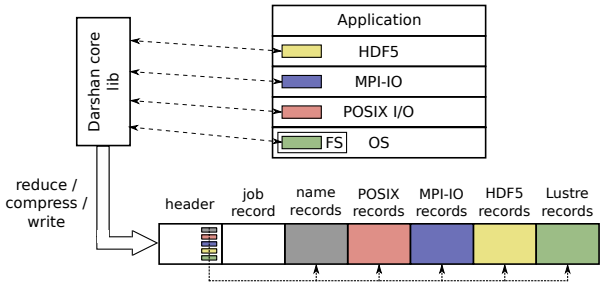


Fig. 1. High-level overview of Darshan's modularized runtime architecture and self-describing log file format.

Figure 1 provides an overview of our design that illustrates the interaction between instrumentation modules, Darshan core, and the output log file format produced. As the application executes, the instrumentation modules generate data records characterizing the application's I/O workload within different components of the I/O stack. The modules register each of these data records with the Darshan core to ensure that the records are persisted in the final output log. At application shutdown time, the Darshan core library provides each module an opportunity to reorganize its data records (e.g., modules can reduce data from records that are common to all processes) before compressing and collectively writing them to the log file. A self-describing file format is employed, wherein a header indexes each module's extent in the log file. This allows log file readers to easily find and extract the various record types using module-specific parsing code.

Instrumentation modules must first register themselves with the Darshan core library to request memory for buffering I/O data records, and this registration process usually occurs the first time the module intercepts a function of interest. Modules typically create one data record for each file accessed by the application, and each record stores module-specific data characterizing the I/O workload to that file. Modules must also register each of these records with Darshan core to obtain a pointer for storing its contents in the module buffer and to allow for other necessary bookkeeping. As part of the record registration process, modules can associate arbitrary-length names with each record, providing a convenient way to store the complete path of the corresponding file. After registering a record with Darshan core, modules may cache the record's buffer address (e.g., in a hash table keyed by file path) so the record can be readily modified by subsequent instrumentation. Data records do not have to map to files, however, and modules are free to populate them at other granularities.

When the Darshan core library initializes itself (by intercepting `MPI_Init()`), it allocates a set of buffers to store various global components of the Darshan log file including the header (which contains an index of all other log file components); the job-level record; and the name record table, which stores the names assigned to data records during registration. A common buffer is used to store all of the modules' records, and this buffer space is allocated to modules in a first-come, first-served manner. The size of this shared buffer is configurable (to cap Darshan's memory usage); but once the buffer space is exhausted, Darshan prevents new modules from reserving any memory. Darshan core also keeps track of the amount of remaining space in each module's buffer so that if a module runs out of buffer space, Darshan core will prevent it from registering new records.

At application shutdown time, Darshan core intercepts `MPI_Finalize()` and collects all the instrumented data to be written to the log file. Rank 0 constructs and writes the header and job-level record, and then all application processes collectively write records on a module-by-module basis. Each module is first given an opportunity to perform any required reorganization of records and internal cleanup through a finalization function that it registered during initialization. After this module-specific finalization completes, Darshan core compresses and collectively appends the module's record buffer to the output log file. The header's index then is updated to indicate where the module's compressed data is located in the log file, and the process continues for the next module.

B. New instrumentation modules

Leveraging the new Darshan concepts outlined in the previous section, we implemented a new set of instrumentation modules characterizing components of the I/O stack that have not been well covered by Darshan in the past. In particular, we describe here the design of a Lustre file system module, a stdio I/O interface module, and a module for collecting IBM Blue Gene/Q–specific parameters.

Lustre module: The modularity of Darshan provides a convenient way to abstract the implementation-specific interfaces of different file systems and gather performance-critical parallelization parameters such as stripe width and size. To demonstrate this, we implemented such a module for the Lustre file system that uses a Lustre-specific `ioctl` interface to retrieve striping information and file system geometry for files that are stored on Lustre. For each such file, the Lustre module stores the stripe size, the stripe width, the total number of object storage targets (OSTs) and metadata targets (MDTs) in the file system, and an enumeration of the OSTs over which the file is striped.

While these file system-specific data obviously provide key details on understanding how application I/O workloads exploit file system parallelism, they can also provide additional context for the measurements made by other instrumentation modules. For example, the combination of a file's stripe size (from the Lustre module) and its most frequent transaction size (from the POSIX module) can quickly indicate mismatches that would result in significant lock contention. Additionally, the enumeration of OSTs can be combined with external system information to identify I/O that may have been impacted by faulty or underperforming object storage servers (OSSes).

stdio module: In previous work [9] we observed that many production applications rely on text-based I/O in leadership-class computing facilities. We therefore implemented a "stdio"

module to more fully characterize the `stdio.h` family of functions, such as `fopen()`, `fprintf()`, and `fscanf()`. These I/O functions are often used in fields such as genomics and biology that store sequencing information in text format. A subset of the stdio interface was previously instrumented in Darshan as part of POSIX characterization, but we elected to split it into a separate module in order to add more detail and coverage without activating the functionality for applications that do not need it. The data recorded in the stdio module is similar to that recorded in the POSIX module and includes data such as operation counts, number of bytes transferred, and amount of time spent in I/O functions.

BG/Q module: While Darshan's primary focus is on instrumenting specific components in the I/O stack, it can also provide users and administrators with data describing how a job interacts with underlying platform. We have developed a module for gathering platform-specific data for IBM BG/Q systems that includes the number of allocated compute nodes, the number of processes per compute node, the number of assigned I/O nodes, and details on the dimensions of the torus network for the job's compute partition. This data is gathered mainly by querying hardware "personality" data using the BG/Q system programming interface.

We note that the BG/Q module is a slight departure from those previously described in that it does not instrument any particular functions and instead just captures a static set of data. Darshan provides special hooks that allow modules like this to capture the data they need during the initialization of the Darshan runtime environment.

C. Robust logging

Previous observations have shown that Darshan's coverage can be reduced by applications that link Darshan in but do not successfully generate logs at application shutdown time. This situation is typically due to applications not calling `MPI_Finalize()`, perhaps because the application crashed or exceeded its wall time limit. We decided to implement a new, robust logging technique in Darshan to combat this issue, ensuring that I/O characterization data is persisted regardless of whether the application terminates properly.

To accomplish this, we leverage the `mmap` system call to memory map Darshan's data structures to log file as the application executes, rather than simply `malloc`'ing memory to store them. The mapped region is configured with the `MAP_SHARED` flag set to ensure that updates to any addresses in the map are (eventually) carried through to the file backing the mapping,[3] protecting against data loss in the event of abnormal termination. To simplify the recovery of Darshan data from these temporary log files, we organize the memory mapping in Darshan's typical log file format, with the caveat that the data is left in uncompressed form. In the expected case that the application terminates normally, Darshan uses its traditional shutdown procedure and simply removes the temporary log files. If the application does terminate unexpectedly and the temporary log files are stored locally on compute nodes, the job scheduler may have to move the log files to a globally visible file system.

This robust logging technique results in a single, uncompressed log file per-process. This data format is neither space-efficient nor compatible with existing analysis tools. To address this, we developed the `darshan-merge` utility for postprocessing these temporary log files and merging them into a single, compressed log file per job. This tool reads in each process's temporary log file and aggregates the data into a single output log, much like the traditional shutdown procedure does. This process of converting the temporary per-process log files into per-job log files could be tightly integrated with the job scheduler on some systems, or it could be handled with a periodic cron job.

[3]To keep Darshan's overheads low, we generally recommend that the backing file be at least node-local, preferably backed by a RAM disk or other low-latency storage.

IV. Performance validation

Since Darshan is intended for deployment in production on large-scale HPC systems, performance and scalability are critical requirements of its design. To validate that our design satisfies these requirements, we investigated both the overhead of Darshan's method for instrumenting I/O functions and the time taken by Darshan to complete its shutdown process and generate an output log file.

Each of these experiments was performed on the Edison system at NERSC, a 2.5 petaflop Cray XC30 system composed of 5,576 compute nodes and 133,824 compute cores (24 compute cores per node). We directed all I/O to the `cscratch` Lustre file system, a global scratch volume shared between Edison and the new Cori system at NERSC, offering over 28 PiB of capacity and a peak performance of over 700 GiB/s. For each experiment, we compared relevant performance metrics across the following Darshan configurations to determine whether any exhibit increased overheads: Darshan 2.3.0, Darshan 3.1.0, and Darshan 3.1.0 configured to use the `mmap`-based logging mechanism (with the backing files stored on the RAM disks on Edison compute nodes).

A. Darshan instrumentation overhead

In order to limit any possible perturbations of application I/O performance, Darshan's instrumentation methods must impose negligible overhead. Previous analysis [7] has demonstrated that instrumentation overheads in prior Darshan versions are essentially negligible, even for latency-bound I/O operations. In the experiments presented here, we are primarily trying to determine whether new Darshan design features have introduced any obvious increases in this overhead. Possible causes for increased overheads in the new Darshan versions include the required coordination between modules and the Darshan core library to register data records and the new `mmap`-based memory allocation mechanism. To ensure these features do not adversely affect application I/O performance, we simply compared the observed I/O time of an application linked with each proposed Darshan configuration.

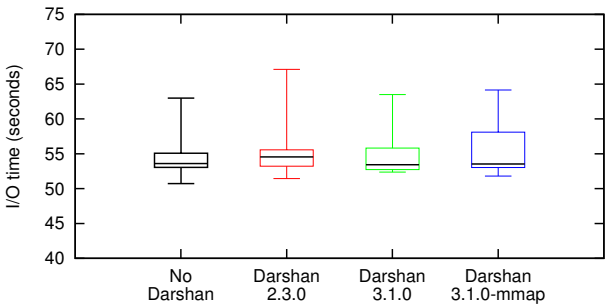


Fig. 2. I/O time reported by an IOR file per-process experiment with and without Darshan instrumentation.

We used the popular IOR benchmark [12] to generate our target I/O workload, with the benchmark configured to use a single file per-process and to use independent MPI-IO operations. Specifically, each of a total of 6,000 application processes (on 250 Edison compute nodes) writes a total of 512 MiB to its own file using 512 KiB accesses. This workload results in an aggregate write size of 3,000 GiB and requires over 12 million total MPI-IO and POSIX I/O operations (over 2,000 operations per process). We built four different IOR executables for these experiments: one with no instrumentation and three linked with each Darshan configuration we outlined previously. To differentiate normal system variance from deviations caused by Darshan, we submitted 15 distinct jobs for each IOR version. We measured the I/O time of each benchmark using the total I/O time reported by IOR, which is a measure of the duration between the time of the first open of an output file on any process to the time of the last close of a file on any process. We focus on one job size for these experiments since the overhead will not change with scale (Darshan avoids I/O and communication within its wrappers, deferring this activity to the shutdown procedure which we analyze in Section IV-B).

In Figure 2, we provide box plots of the results of these tests for each IOR version. These plots give the minimum, median, and maximum results, as well as the first and third quartiles, to indicate the distribution of I/O times in each case. We observe that Darshan appears to have little impact on the I/O performance of the benchmark, regardless of the configuration that was used. The I/O time distributions are comparable in each case, and each of the IOR versions linked with Darshan experiences less than 2% increase in median I/O time, which is nearly negligible when considering typical variability of the file system. We conclude that Darshan's instrumentation methods introduce minimal (if any) measurable overhead in I/O performance.

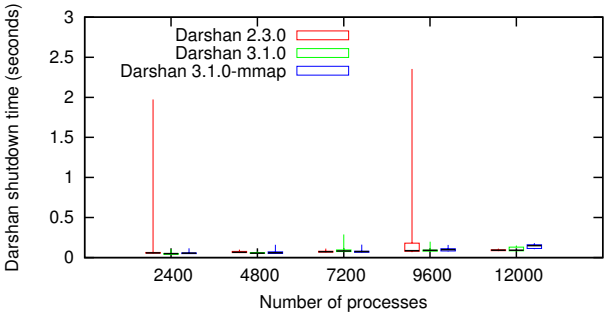
B. Darshan shutdown overhead

The Darshan shutdown process involves aggregating output data records across all processes, compressing these records, and collectively writing the records out to the Darshan log file. This process is invoked by Darshan after the application has finished processing by intercepting `MPI_Finalize()`. While this overhead is essentially transparent to applications, it is still important that the process complete efficiently in order to allow job resources to be reclaimed quickly by the scheduler. Although this shutdown process involves the same general steps (reduce, compress, write) for each Darshan version, we note that the Darshan 3.1.0 versions perform these steps on a per-module basis rather than once globally. The primary aim of these experiments is to discern whether Darshan's new modularized architecture causes an excessive increase in shutdown overhead.

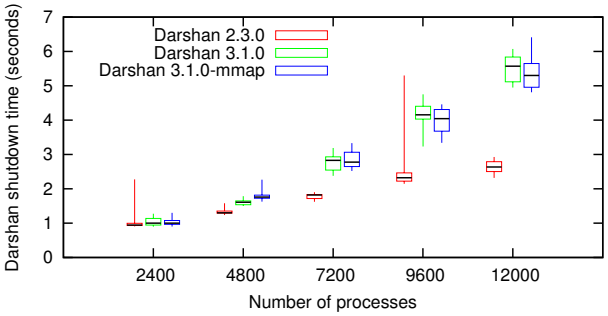
We measured the performance of the shutdown process for each Darshan configuration using a low-level benchmark that injects synthetic data records corresponding to different I/O workloads into the Darshan runtime environment. We then invoked the Darshan shutdown procedure and instrumented the total time taken to complete. It is important to note that we are measuring the traditional shutdown procedure even in the mmap case. This is the preferred output method for all applications that complete successfully; there is no need to merge and reconstitute intermediate output as a post-processing step unless the application crashes. For each workload and each Darshan version, we collected 10 independent benchmark samples to obtain a distribution of shutdown times. We also executed the benchmark at numerous job sizes, ranging from 2,400 application processes up to 12,000 processes, to evaluate how the shutdown process scales in each version. The synthesized workloads include a single shared file approach and a file per-process approach, each using both the MPI-IO and POSIX interfaces. These workloads are representative of the checkpoint/restart I/O strategy that is commonly employed in HPC applications.

Figure 3 provides the shutdown benchmark performance results for each Darshan version. For shared file workloads, Darshan uses MPI collective communication to determine which records are shared globally and to reduce each of these shared records into a single aggregate record. Rank 0 ends up with the reduced aggregate records and is responsible for writing them to the log file. In Figure 3(a), we provide the shutdown overhead results for a single shared file workload. The figure shows little difference in the observed shutdown times for each configuration, with most samples taking around 100 milliseconds to shut down. This overhead remains mostly constant across all scales, save for a couple of outlier results. These results indicate that both Darshan 3.1.0 versions attain comparable performance to Darshan 2.3.0 for a shared file workload, despite the added complexity in the shutdown process.

The shutdown overhead results for a file per-process workload are given in Figure 3(b). Unlike the shared file examples, Darshan does not perform reductions of shared file records in this case; instead it uses collective I/O to write out each process's unique file record. As expected, the shutdown times for these workloads scale linearly with the job size. Each of the Darshan 3.1.0 versions exhibits slightly longer shutdown time than Darshan 2.3.0 at smaller scales, with the disparity in



(a) 1 shared file



(b) 1 file per-process

Fig. 3. Observed shutdown time for each Darshan version when instrumenting common HPC I/O workloads.

shutdown time increasing with the job size. This is an artifact of the modularized log format in this version: the decision to compress and write each module's data independently leads to reduced compression efficiency and more write operations, causing a clear increase in shutdown cost. More details on the reasons behind reduced shutdown performance in the modularized version of Darshan are provided in an earlier technical report [13]. Nevertheless, each Darshan version still aggregates, compresses, and writes its recorded data in less than 7 seconds for all configurations evaluated here. For jobs that run for hours (which is commonly the case on production HPC systems), this shutdown time is essentially negligible.

V. Case Studies

Next, we consider case studies where Darshan instrumentation can provide application users and system administrators alike valuable insight into the performance of I/O workloads of interest. In particular, we utilize Darshan's newly developed instrumentation modules to determine whether they enable increased intuition into the behavior of representative HPC application I/O workloads.

A. HACC-IO

HACC (Hardware/Hybrid Accelerated Cosmology Code) is an extreme-scale cosmology framework that uses particle-mesh techniques to simulate the evolution of the universe [14]. The I/O kernel of HACC has been released as a stand-alone benchmark, called HACC-IO, that is a part of the CORAL benchmark suite.[4] Since this benchmark suite was designed to be representative of U.S. Department of Energy applications, we chose to use HACC-IO as an exemplar I/O workload to demonstrate the utility of the Lustre module when analyzing the performance characteristics of HPC storage systems.

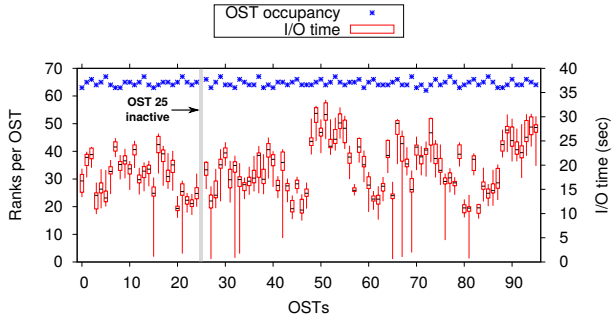
We modified the HACC-IO benchmark to perform only the checkpoint (write) phase of the workload and to skip the restart/validate (read) phase. We configured HACC-IO to utilize the POSIX interface and generate a single output file per MPI process, with each process writing over 2 million particles, or 100 MiB of data. We also modified the benchmark to use the `O_DIRECT` and `O_SYNC` flags when opening checkpoint files to ensure the I/O operations would not be served by cache. This experiment was run on Edison using 6,144 application processes on 256 compute nodes, resulting in an aggregate workload write size of 600 GiB. Checkpoint files were written to Edison's Lustre-based `scratch1` file system, which is composed of 24 Lustre OSSes, each with 4 OSTs (96 OSTs total), and is capable of 48 GiB/s peak write performance. We configured our HACC-IO output directory to have a stripe width of 1, resulting in each process's output file mapping to a single OST.

Figure 4 presents the performance of the HACC I/O workload using data from Darshan's Lustre module to frame the results from the perspective of the underlying file system. Figure 4(a) illustrates the total number of ranks whose output maps to each OST (which we term the *occupancy*) and the distribution of I/O times for processes utilizing that OST. The data demonstrates uniform occupancy, indicating that the file system effectively balances the distribution of application processes' output files across all available OSTs for this run. Although this should lead to similarly well-balanced I/O time distributions across all OSTs, this is not the case; rather, Figure 4(a) reveals distinct blocks of contiguous OSTs (e.g., OSTs 48–55 and 88–95) that exhibit consistently longer I/O times.

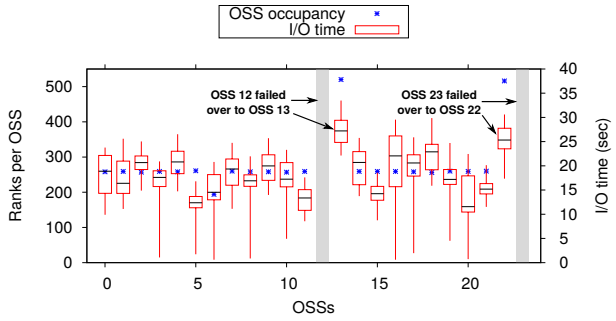
To determine the root cause of these blocks of slower OSTs, we mapped each OST to the IP address of the OSS serving it. The aggregate file I/O times at this per-OSS level, shown in Figure 4(b), more clearly highlight the relationship between the anomalously slow OSTs. Contrary to our expectation that each contiguous group of 4 OSTs would map to a distinct OSS address (e.g., OSTs 0–3 to OSS 0, OSTs 4–7 to OSS 1), the two blocks of eight slow OSTs each map to a single OSS with twice as many OSTs as their higher-performing peers.

After identifying these two oversubscribed OSSes, we were able to confirm with NERSC systems engineers that these two OSSes were indeed serving twice as many OSTs as a result of an earlier system failure. The underlying Lustre storage appliances on Edison implement an active-active failover strategy, allowing each OSS to take over the OSTs of its partner OSS in case of a failure. This failover process maintains data availability in the case of OSS failure, but it clearly results in reduced I/O performance of affected OSSs because of the

[4] https://asc.llnl.gov/CORAL-benchmarks/



(a) per-OST I/O performance



(b) per-OSS I/O performance

Fig. 4. Performance of HACC-IO writing a checkpoint of 12.9 billion particles.

increased load associated with servicing the I/O requests of twice as many OSTs.

This case study demonstrates how aggregated Darshan log data can be used to identify systemwide I/O problems that would be otherwise opaque to users. In the case of the HACC-IO checkpoint workload presented here, overloaded Lustre OSSes resulted in a 13% increase in I/O time. While this reduced I/O performance would likely be obvious to an application user, determining the underlying root cause of the issue would be difficult without detailed instrumentation of the different components in the I/O stack. This could cause application developers or users to waste valuable resources debugging performance problems that are caused by general performance degradation in the system itself, for instance.

B. HMMER

We selected the HMMER [15] application as a case study for stdio characterization. HMMER is a biology application that searches genome sequence databases using hidden Markov models (HMMs). HMMER uses stdio functions exclusively for reading and writing text data. Our example configuration used HMMER version 3.1b2 compiled with MPI support using the Intel C compiler. We executed `hmmsearch` with 4 processes on a single node of Edison using the following command line arguments: `--mpi --noali --notextw <HMM file> <sequence DB>`. We used the `Pfam-A.hmm` HMM file (provided by the Pfam Consortium) and `uniprot_sprot.fasta` sequence DB file

TABLE I
PER-FILE I/O CHARACTERISTICS FOR THREE HMMER CONFIGURATIONS

	config A	config B	config C
	HMM (input)		
file size	1.3 GiB	1.3 GiB	1.3 GiB
I/O volume	5.0 GiB	5.0 GiB	5.0 GiB
I/O time	12.5 s	12.4 s	12.1 s
I/O count	36.2 M	36.2 M	36.2 M
seek count	0	0	0
	Sequence DB (input)		
file size	253.4 MiB	253.4 MiB	253.4 MiB
I/O volume	4.0 TiB	4.0 TiB	4.0 TiB
I/O time	710.8 s	708.4 s	390.8 s
I/O count	1.1 B	1.1 B	1.1 B
seek count	8.3 M	8.3 M	8.3 M
	output		
file size	1.3 GiB	1.3 GiB	1.3 GiB
I/O volume	1.3 GiB	1.3 GiB	1.3 GiB
I/O time	155.8 s	16.6 s	17.9 s
I/O count	16.1 M	16.1 M	16.1 M
seek count	0	0	0
	totals		
file size	2.8 GiB	2.8 GiB	2.8 GiB
I/O time	**879.1 s**	**737.4 s**	**420.8 s**
I/O volume	4.0 TiB	4.0 TiB	4.0 TiB
I/O count	1.1 B	1.1 B	1.1 B
seek count	8.3 M	8.3 M	8.3 M

(provided by the UniProt Consortium) as example input files. Input and output files for these experiments were again stored on the `cscratch` Lustre file system available to Edison users.

Table I summarizes Darshan characterization data for each file opened by the `hmmsearch` application. A total summation is shown in the bottom portion of the table. We examined the following distinct I/O configurations. "A" corresponds to a conventional configuration using the command line arguments shown above. "B" adds a `-o <output file>` argument to explicitly write output to a file rather than `stdout`. "C" is the same as B, except that the sequence DB file was placed in a RAM disk on the compute node before execution.

In all three cases we see that I/O time is dominated by access to the sequence database file. Although this file is only 253 MiB, the application issues over 8 million seek operations and 1 billion read operations to transfer a total of 4 TiB of data from the file. The application benefits from significant client-side data caching in this case because all four processes are located on the same physical compute node, but this is still a challenging workload for most parallel file systems. Configuration C takes advantage of the relatively small total size of the file to enact a simple optimization: the file is copied to a local RAM disk on the compute node (an operation that takes less than a second) prior to execution. This eliminates over 300 seconds of I/O time during application execution. We also investigated the impact of two different data output strategies in this case study. Configuration A uses the default hmmsearch output method in which all results are written to `stdout`. The batch environment on Edison redirects this output to a text file in the user's home directory. Configuration B explicitly writes results to a text file on the parallel file

system. In both cases all data is written by rank 0 in the application using the same number of system calls, but the latter configuration is nearly 10 times faster.

Overall, the Darshan stdio module allows us to observe key I/O behavior for this application and understand the impact of various tuning strategies. In particular, a surprising amount of redundant read activity is noted in one of the input files (which is more suited to low-latency local storage access than high-latency remote storage access), and output performance varies significantly depending on the storage target. These phenomena are likely to play a more prominent role in larger-scale application executions.

VI. Related Work

The HPC community has done a great deal of research regarding I/O tracing tools, including IPM [16], mpiP [17], //TRACE [18], ScalaIOTrace [19], and Recorder [20]. Of these tools, only ScalaIOTrace and Recorder explicitly trace data at multiple layers of the I/O stack (i.e., the POSIX and MPI-IO layers, with Recorder collecting additional HDF5 trace data as well). These tools are not generally deployed in production because of the high computational and storage overheads associated with gathering detailed information on each I/O operation issued by an application. Further, they typically rely on some type of postprocessing to distill actionable tuning decisions out of the traces.

The Charisma [21] project was started in 1993 to characterize the I/O workloads of multiprocessor scientific codes running on two different production systems using I/O tracing. The primary aim of this research was to compare the types of workloads present on each system in order to determine common I/O trends and, ideally, to guide future file system designs. This research was highly influential to Darshan and other subsequent I/O characterization tools proposed by the HPC community. The work in this paper clearly extends the scope of I/O workload characterization beyond the level of file system reads and writes, instead encompassing numerous layers and components in the I/O stack.

SIOX [22] and IOPin [23] are examples of related research that characterize HPC I/O workloads across multiple layers of the I/O stack. In fact, each of these works extend the application-level I/O instrumentation approach that Darshan uses to also account for other remote (i.e., not compute node local) components in the I/O stack. For instance, SIOX instrumentation encompasses network interfaces, file system servers, and file system storage devices, as well as application-level instrumentation. Thus, the SIOX framework can provide fine-grained details on individual I/O operations and determine causal relationships of operations across components of the I/O stack. IOPin uses dynamic instrumentation methods to trace I/O workloads from the application layer all the way to storage servers to determine end-to-end performance characteristics. The IOPin methods appear to be specifically tied to the PVFS file system, however. Also, while SIOX and IOPin can provide valuable insights into the performance of the I/O critical path for target workloads, they exhibit too much overhead to run full-time in production.

Petesch and Swan [24] present work correlating application-level I/O instrumentation with file system specific information to pinpoint I/O issues within Lustre. Specifically, the authors manually instrument the IOR benchmark and manually gather Lustre OST mapping data to analyze system I/O performance and isolate performance issues to specific Lustre components, such as spinning disks or LNET routers. The resulting analysis is therefore relevant only to the Lustre file system and to I/O workloads that can be reconstructed by using IOR, although the applied principles are generalizable to other contexts.

VII. Conclusions

As the complexity of large-scale HPC I/O subsystems continues to grow, application users and system administrators are finding increasing difficulty reasoning about the I/O performance of their workloads. While I/O profiling tools have traditionally proved helpful in analyzing and tuning application workloads, these tools exhibit inflexible designs that are ill-suited for increasingly complex HPC environments. In this work, we have proposed a new, modularized design for the Darshan I/O characterization tool to allow it to be easily extended to account for new components in the HPC I/O stack. We have evaluated the overheads of the new design and have found that it is highly scalable and thus amenable to full-time deployment in production. We have also shown how Darshan can provide both application users and system administrators valuable insights into application I/O workload behavior.

We will continue to investigate and develop new Darshan instrumentation modules that can provide intuition into I/O workload performance on HPC systems. These modules may allow for the instrumentation of experimental I/O interfaces or emerging storage hardware components, for instance. We are also interested in correlating Darshan's application view of I/O behavior with I/O data instrumented from other relevant sources throughout the system (e.g., network interconnects, burst buffers, storage servers) to enable a more comprehensive understanding of I/O performance across HPC systems.

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] R. Latham, C. Daley, W.-k. Liao, K. Gao, R. Ross, A. Dubey, and A. Choudhary, "A case study for scientific I/O: improving the FLASH astrophysics code," *Computational Science & Discovery*, vol. 5, no. 1, p. 015001, 2012.

[2] V. Vishwanath, M. Hereld, and M. E. Papka, "Toward simulation-time data analysis and I/O acceleration on leadership-class systems," in *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*. IEEE, 2011, pp. 9–14.

[3] J. M. Dennis, J. Edwards, R. Loy, R. Jacob, A. A. Mirin, A. P. Craig, and M. Vertenstein, "An application-level parallel I/O library for Earth system models," *International Journal of High Performance Computing Applications*, p. 1094342011428143, 2011.
[4] W. Bhimji, D. Bard, M. Romanus, D. Paul, A. Ovsyannikov, B. Friesen, M. Bryson, J. Correa, G. K. Lockwood, V. Tsulaia *et al.*, "Accelerating science with the NERSC burst buffer early user program."
[5] 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*. ACM, 2009, p. 40.
[6] W. Yu, J. S. Vetter, and H. S. Oral, "Performance characterization and optimization of parallel I/O on the Cray XT," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–11.
[7] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 characterization of petascale I/O workloads," in *IEEE International Conference on Cluster Computing and Workshops, 2009. CLUSTER'09*. IEEE, 2009, pp. 1–10.
[8] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and improving computational science storage access through continuous characterization," *ACM Transactions on Storage (TOS)*, vol. 7, no. 3, pp. 1–14, 2011.
[9] 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*. ACM, 2015, pp. 33–44.
[10] S. Snyder, P. Carns, R. Latham, M. Mubarak, R. Ross, C. Carothers, B. Behzad, H. V. T. Luu, S. Byna *et al.*, "Techniques for modeling large-scale HPC I/O workloads," in *Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems*. ACM, 2015, p. 5.
[11] M. Fahey, N. Jones, B. Hadri, and B. Hitchcock, "The automatic library tracking database," *Proceedings of the Cray User Group*, 2010.
[12] Lawrence Livermore National Laboratory, "IOR benchmark," https://github.com/chaos/ior, 2015.
[13] S. Snyder, P. Carns, K. Harms, R. Latham, and R. Ross, "Performance evaluation of Darshan 3.0.0 on the Cray XC30," Argonne National Laboratory (ANL), Tech. Rep., 2016.
[14] S. Habib, V. Morozov, H. Finkel, A. Pope, K. Heitmann, K. Kumaran, T. Peterka, J. Insley, D. Daniel, P. Fasel *et al.*, "The universe at extreme scale: multi-petaflop sky simulation on the BG/Q," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 4.
[15] S. R. Eddy and T. J. Wheeler, "HMMER: biosequence analysis using profile hidden Markov models," http://www.hmmer.org/, 2016.
[16] A. Uselton, M. Howison, N. J. Wright, D. Skinner, N. Keen, J. Shalf, K. L. Karavanic, and L. Oliker, "Parallel I/O performance: from events to ensembles," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–11.
[17] J. Vetter and C. Chambreau, "mpiP: lightweight, scalable MPI profiling," *http://mpip.sourceforge.net*, 2004.
[18] M. P. Mesnier, M. Wachs, R. R. Simbasivan, J. Lopez, J. Hendricks, G. R. Ganger, and D. R. O'Hallaron, "//TRACE: parallel trace replay with approximate causal events." USENIX, 2007.
[19] 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*. ACM, 2009, pp. 26–31.
[20] 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)*. IEEE, 2013, pp. 1–5.
[21] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. S. Ellis, and M. L. Best, "File-access characteristics of parallel scientific workloads," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 10, pp. 1075–1089, 1996.
[22] M. C. Wiedemann, J. M. Kunkel, M. Zimmer, T. Ludwig, M. Resch, T. Bönisch, X. Wang, A. Chut, A. Aguilera, W. E. Nagel *et al.*, "Towards I/O analysis of HPC systems and a generic architecture to collect access patterns," *Computer Science-Research and Development*, vol. 28, no. 2-3, pp. 241–251, 2013.
[23] S. J. Kim, S. W. Son, W.-k. Liao, M. Kandemir, R. Thakur, and A. Choudhary, "IOPin: runtime profiling of parallel I/O in HPC systems," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*. IEEE, 2012, pp. 18–23.
[24] D. Petesch and M. Swan, "Instrumenting IOR to diagnose performance issues on Lustre file systems," *Proc. Cray User Group*, 2013.