Production I/O Characterization on the Cray XE6

Philip Carns,* Yushu Yao,† Kevin Harms,* Robert Latham,* Robert Ross,* Katie Antypas†

*Argonne National Laboratory, Argonne, IL, USA, {carns,robl,rross}@mcs.anl.gov, harms@alcf.anl.gov
†Lawrence Berkeley National Laboratory, Berkeley, CA, USA, {yyao,kantypas}@lbl.gov

***Abstract*—I/O performance is an increasingly important factor in the productivity of large-scale HPC systems such as Hopper, a 153,216 core Cray XE6 system operated by the National Energy Research Scientific Computing Center. The scientific workload diversity of such systems presents a challenge for I/O performance tuning, however. Applications vary in terms of data volume, I/O strategy, and access method, making it difficult to consistently evaluate and enhance their I/O performance.**

We have adapted the Darshan I/O characterization tool for use on Hopper in order to address this challenge. Darshan is an I/O instrumentation library that collects I/O access pattern information from large-scale production applications with minimal overhead. In this paper we present our experiences in deploying Darshan on the Cray XE6 platform, including performance evaluation of Darshan with up to 98,304 processes and a case study of how to identify applications that can benefit most from I/O performance tuning. Darshan was automatically enabled for all Hopper users in November 2012 and instruments over 5,000 jobs per day as of April 2013.

I. Introduction

I/O performance is an increasingly important factor in the productivity of large-scale HPC systems. I/O performance is difficult to consistently quantify and understand across applications, however. This situation is especially for systems that are used for research in multiple disciplines, such as the Hopper system at the National Energy Research Scientific Computing Center (NERSC). Hopper is used by scientists from a wide array of fields, including physics, materials, biology, and climate. Applications from these fields vary not only in the type of data they ingest and generate, but also in the strategy used to access that data. Instrumenting and tuning specific applications or benchmarks in this environment is an important activity, but it provides limited insight across application domains. The most effective general way to analyze the I/O behavior of such a diverse collection of applications is through the use of lightweight instrumentation tools that can be deployed to automatically collect data from a wide cross-section of production jobs.

Darshan is an application-level I/O characterization library that has been designed to address this need [1]. Darshan instruments I/O function calls at the POSIX, MPI-IO, PnetCDF, and HDF5 level in order to record concise access pattern information for every file opened by an application. Data is collected independently at each MPI process using a bounded amount of memory. When the application shuts down, Darshan aggregates the data, compresses it in parallel, and writes it to a single binary-format output file. The resulting output file does not contain a complete trace of I/O activity from the job. It instead records a set of counters, cumulative timers, and histograms for each file that summarize the application's access pattern. This data can describe a wide range of I/O behavior, including the number of files opened, the amount of data read and written, common access sizes, strided access patterns, and time spent performing different types of I/O operations. In previous work we have shown how this data can be used to perform broad, ongoing analysis of system I/O behavior [2], and the Argonne Leadership Computing Facility has made a collection of anonymized Darshan logs available for public analysis [3]. Darshan has been used to provide data for automatic tuning efforts as well [4].

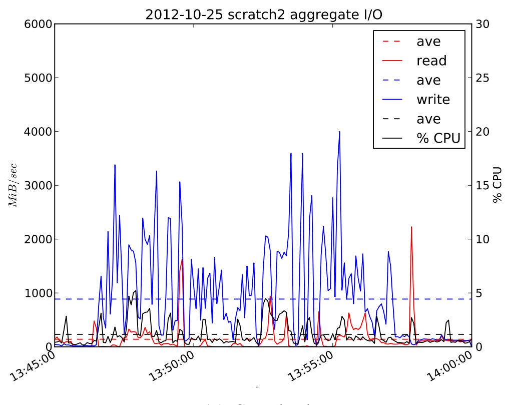
In this work we explore the deployment of Darshan on Hopper, a 1.28-petaflop Cray XE6 system operated by NERSC. NERSC provides access to Hopper through multiple allocation programs in order to support Department of Energy research as well as the NERSC Initiative for Scientific Exploration and educational programs. Hopper is used by thousands of researchers spanning an assortment of scientific domains. Darshan was evaluated on Cray XE6 systems in a limited fashion in previous work [5], but that study focused a small set of workloads without full-scale deployment, tuning, or application case studies.

Although the Darshan library was designed for portability, we faced a number of challenges in adapting and tuning Darshan for the Cray XE6 environment on Hopper. In this work we report on our experiences in preparing Darshan for use in this environment. We evaluate the performance of Darshan to confirm that it is suitable for full-time deployment on large-scale Cray systems. We also explore how data collected with Darshan can be used to identify candidate applications that can benefit most from additional I/O tuning.

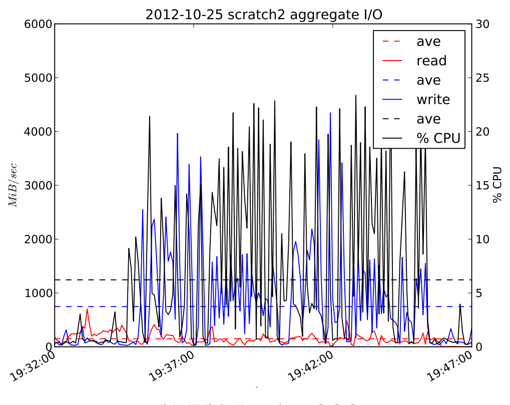
II. Test environment

All experiments and case studies in this work were carried out on the Hopper Cray XE6 system. Hopper contains 153,216 compute cores and 217 TiB of memory. Multiple file systems are available to users on Hopper, including a home file system, two high-performance scratch file systems, a global scratch file system (visible across multiple computer systems), and a project data file system. Unless otherwise noted, all Darshan measurements were taken on the second scratch volume (SCRATCH2), a 1 PiB Lustre file system with 26 I/O servers (OSSs) and 156 object storage targets (OSTs).

At the time of writing Hopper used version 5.16 of the Cray Programming Environment (`xt-asyncpe-5.16`). We used



(a) Standard



(b) With Darshan 2.2.3

Fig. 1. Lustre server CPU utilization and I/O throughput as reported by the Lustre Monitoring Tool (LMT) for an example application accessing large shared files with Darshan version 2.2.3 in preliminary testing.

this environment unmodified in this study with the exception that we elected to use `cray-mpich2-5.5.5` rather than the default `cray-mpich2-5.6.0` MPI implementation. We are currently working with Cray Inc. to evaluate and address a potential performance regression in 5.6.0 that affects an MPI-IO collective access pattern generated by the Darshan library. We used version 2.2.6-pre1 of the Darshan library for all experiments unless otherwise noted.

III. Cray Platform Challenges

The Cray Programming Environment exhibits the following notable characteristics from a Darshan instrumentation perspective:

- Support for PGI, Pathscale, Cray, Intel, and GNU compilers
- Applications statically linked by default, though dynamic linking is also supported
- Compilation invoked through a set of unified compiler scripts (`cc`, `ftn`, and `CC`)
- Integrated support for PnetCDF and HDF5 high-level I/O libraries

In order to ensure the most coverage, we focused on supporting static compilation using each of the five supported compiler suites. The static instrumentation method used by Darshan relies on linker function wrapping options and the MPI profiling interface (PMPI), both of which require link-time instrumentation. We used customized compiler scripts to insert instrumentation for the experiments performed in this paper, but future versions of Darshan will leverage the Cray `PE_PRODUCT_LIST` mechanism instead. Software modules can use the `PE_PRODUCT_LIST` environment variable to specify additional linker and library options to be used by the standard Cray compiler scripts at compile time.

A. Lustre file system tuning

The initial integration tests of Darshan on Hopper were carried out with Darshan version 2.2.3. Using this version of Darshan, we discovered that a subset of applications that used large shared files was suffering from a performance degradation that had not been evident in previous studies of applications that used separate files per process [5]. We used the Lustre Monitoring Tool [6] to further diagnose the problem. Figure 1 shows the CPU utilization of the Hopper Lustre servers over time during the execution of an affected application both with and without Darshan instrumentation. This server-side CPU utilization indicated that the problem was not limited to the compute node environment but that Darshan was also causing an additional workload for the file system itself.

The source of the problem proved to be that Darshan was issuing an extra `stat()` system call each time a file was opened for the first time on each process. This system call served three purposes in Darshan: detecting the optimal transfer size for the file (usually the stripe size in a parallel file system), detecting the device ID for the mounted file system (so that the file record could be easily matched against mounted file systems), and detecting the initial size of the file at open time. The cost of a `stat()` system call varies depending on the file system and the nature of the file being queried, however. On Hopper, a `stat()` call to a widely

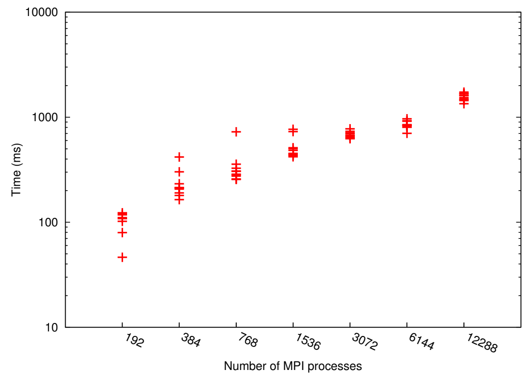


Fig. 2. Time required to stat a widely striped 1 GiB file from all processes simultaneously. There are ten samples at each data point.

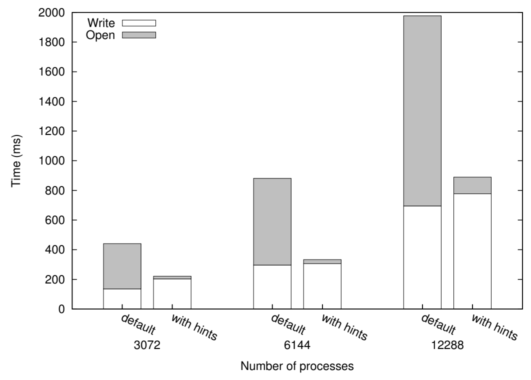


Fig. 3. Time required to open and collectively write 700 bytes from each process with MPI-IO on cray-mpich2 version 5.5.5 as the number of MPI processes is increased. This graph shows average values over 10 runs for each case. The "with hints" experiments specified MPI-IO hints `romio_no_indep_rw=true` and `cb_nodes=4`.

striped file requires contacting every file server in order to calculate the current file size. If a large number of processes do this simultaneously, it can cause a CPU spike on the file servers and detract from application performance. Figure 2 confirms the application-level impact of this behavior by showing the results of a microbenchmark that measures the time required to concurrently stat a widely striped 1 GiB file as the number of application processes on Hopper was varied from 192 to 12,288. This experiment shows that the additional stat traffic implicitly generated by Darshan version 2.2.3 would add up to 1.7 seconds to the time required to open a file with 12,288 processes.

To address this problem, we modified Darshan to avoid issuing implicit `stat()` system calls on newly opened files as of Darshan version 2.2.4. File records are matched to the appropriate mount point according to path rather than device ID. The optimal transfer size for each file is estimated based on either the type of file system or the results of a `fstatfs()` call on the mount point. In future work we intend to investigate the use of the Lustre library API and other file-system-specific APIs for this purpose. The proposed `xstat()` system call in Linux is also a promising solution, but it is not widely available at this time. Darshan no longer records the size of files at open time because there is currently no portably scalable replacement for this functionality.

B. MPI-IO aggregation

The final component of Darshan that we tuned for use on Hopper was the mechanism used to write Darshan logs to disk. Darshan uses MPI-IO and collective write operations for this purpose. Because Darshan results are typically small (sometimes only a few hundred bytes per process, even with unique files on each process), the collective write call benefits considerably from the MPI-IO collective buffering optimization [7]. The Lustre-optimized MPI-IO driver also ensures that collectively buffered writes are stripe-aligned to further improve performance [8]. There may still be substantial cost to simply opening a file on a large number of processes, however, regardless of how the data is subsequently written. To combat this, we took advantage of a Darshan features that allows additional hints to be specified for the collective I/O step, either at compile time (via the `--with-log-hints` configure option) or at run time (via the `DARSHAN_LOGHINTS` environment variable). The most critical hint to specify in this case is `romio_no_indep_rw`, which alerts the MPI-IO layer that no independent I/O will be performed on the file handle. In this scenario, MPI-IO can limit the number of processes that will open a file to only those that will act as aggregators for collective buffering. We further use the `cb_nodes` hint to limit the total number of aggregators to no more than 4, regardless of the number of processes in the application. The outcome of these optimizations can be seen the microbenchmark results shown in Figure 3. This test measured the amount of time required to collectively open and then collectively write 700 bytes from each process. The additional hints reduce the combined cost of these two steps by over 50% at 12,288 processes.

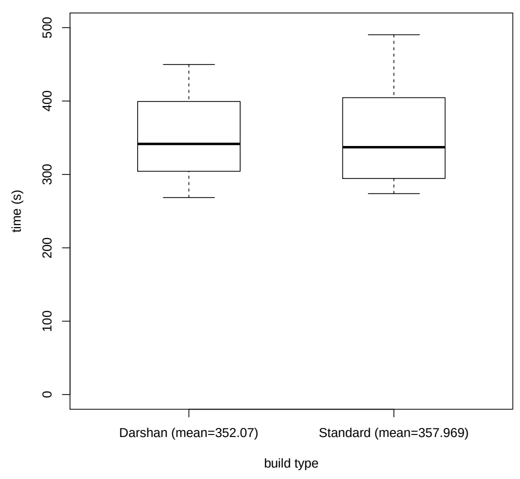
IV. Performance evaluation

The preceding section described how Darshan was adapted and tuned in order to integrate with the Cray Programming Environment, efficiently collect data at file open time, and efficiently write results at application exit time. In this section we evaluate the impact of these optimizations by studying the end-to-end performance of Darshan for representative application workloads.

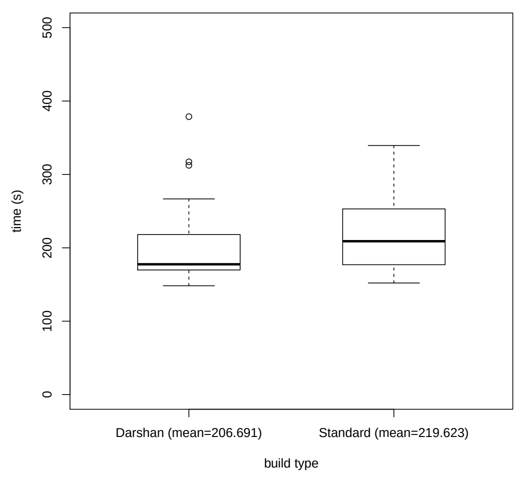
A. End-to-end overhead

Our first goal in performance evaluation was to measure the end-to-end execution-time overhead introduced by Darshan, including any startup costs, instrumentation costs, and log output costs. We chose to use version 2.10.3 of the IOR benchmark for this purpose. IOR is a synthetic benchmark developed by Lawrence Livermore National Laboratory that can be configured to emulate a variety of workloads [9]. We configured IOR to write and read a total of 1.5 TiB of data using 128 KiB MPI-IO access sizes from 12,288 processes. Darshan instrumented both the MPI-IO and POSIX API levels in this example. We tested two variations: a file-per-process configuration and a shared-file configuration. The file-per-process configuration utilized the default Lustre settings on Hopper, which striped each file across 2 OSTs. The shared-file configuration used interleaved collective I/O to access files that were striped across all 156 OSTs on the system. In both configurations we used the `reordertasksconstant=1` option in IOR to force each process to read data generated by a different process.

Previous studies have indicated that storage systems at this scale may experience a high degree of performance variance across runs [2], [10], [11]. Variance can stem from a variety of sources, including hardware performance fluctuations, interprocess contention, and interjob contention. These fluctuations make it difficult to isolate the overhead of lightweight I/O instrumentation tools in the context of a production system.



(a) shared-file mode



(b) file-per-process mode

Fig. 4. Execution time of IOR benchmark with 12,288 processes. IOR was configured to read and write a total of 1.5 TiB of data using MPI-IO operations with an access size of 128 KiB in both cases. The shared-file example used an interleaved access pattern with collective I/O and file striping set to use all 156 OSTs. Each box plot summarizes 20 independent samples.

We collected multiple independent samples in an attempt to mitigate this effect. We submitted 20 independent jobs in each of four sample groups: IOR shared files with Darshan, IOR shared files without Darshan, IOR file-per-process with Darshan, and IOR file-per-process without Darshan. We alternated between all four configurations to submit a total of 80 independent jobs between March 14 and March 18, 2013. The experimental jobs never overlapped with each other, but we had no control over the workload generated by other users on the system. For each job we collected the run time of the IOR executable by timing the `aprun` command within the job script. This ensured that we captured the entire benchmark execution time including startup and shutdown but not including job scheduling or node preparation overhead.

Figure 4(a) summarizes the results of this experiment in the IOR shared file, collective I/O case using a box plot. Each box shows the minimum, median, maximum, and Q1 and Q3 quartiles across 20 sample measurements. In this configuration there were no outliers, but there is a wide variation from 268 seconds to 490 seconds for the minimum and maximum times across all data points, meaning that the slowest IOR run was nearly 83% slower than the fastest IOR run. We observed similar variability whether or not Darshan was used.

Figure 4(b) summarizes the IOR execution time in the file-per-process configuration. In this case, the IOR runs that used Darshan appear to exhibit lower Q3 and maximum values, but there were three distinct outliers in that sample set with a value greater than $Q3 + 1.5 * IQR$. The fastest and slowest runs were 148 and 379 seconds, respectively, meaning that the slowest execution time was over 2.5 times slower than the fastest execution time.

Variability in I/O performance makes it difficult to draw any conclusions about overhead from the box plots themselves.

TABLE 1
95% CONFIDENCE INTERVAL FOR THE DIFFERENCE IN MEAN IOR EXECUTION TIME BETWEEN SAMPLES THAT USED DARSHAN AND SAMPLES THAT DID NOT

Benchmark Type	Lower Range (s)	Upper Range (s)
IOR shared file	-42.64472	42.44177
IOR file-per-process	-18.27367	57.31462

We therefore used a Welch Two Sample t-test as computed with R [12] to evaluate the difference in means between the Darshan and non-Darshan examples for both the IOR shared file and IOR file-per-process configurations. The 95% confidence interval for the difference in means in each case is shown in Table 1. There is not sufficient evidence in either case to conclude that Darshan caused a change in the mean performance of IOR in this context.

We note, however, that the IOR file-per-process configuration tends to perform significantly better than the IOR shared-file configuration on Hopper at 12,288 processes. The file-per-process configuration exhibits a higher degree of variability, however.

B. Shutdown time

In the previous section we were unable to measure Darshan overhead in the context of end-to-end execution time due to I/O performance variability. We can use microbenchmarks to investigate the performance of specific mechanisms within the Darshan library independently, however. From previous studies [1] we expect the overhead of instrumenting individual I/O functions to be trivial relative to the cost of file system access, and in Section III-A we eliminated `stat()` system call overhead produced at `open()` time. The most significant remaining source of potential overhead is the shutdown cost

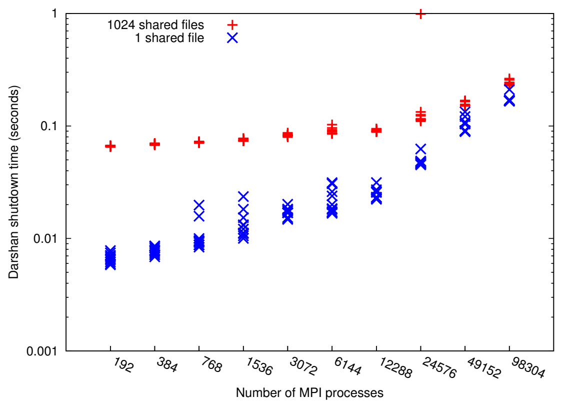


Fig. 5. Darshan data aggregation, compression, and output time for jobs using shared files. Ten samples are shown for each job size.

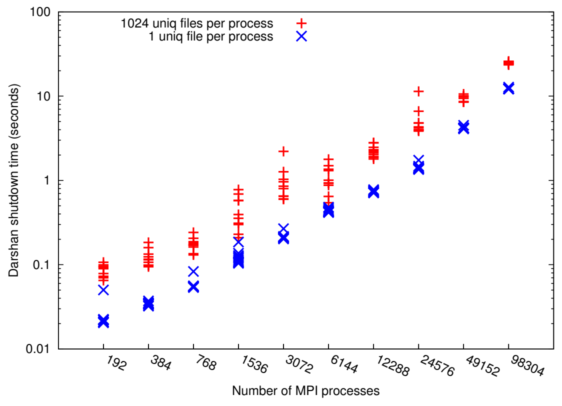


Fig. 6. Darshan data aggregation, compression, and output time for jobs using unique files on each process. Ten samples are shown for each job size.

associated with Darshan.

Darshan intercepts the `MPI_Finalize()` function call in order to collect characterization results from all processes. At this time, it performs a sequence of collective operations to reduce the volume of data (by identifying and merging shared files), compresses remaining data in parallel using `zlib`, and writes the resulting data to a single shared file using MPI-IO. We have constructed a microbenchmark called `darshan-shutdown-bench` to measure the cost of these steps. This microbenchmark does not actually open any any files at run time. It instead populates the Darshan instrumentation system with synthetic file records at each process to simulate various file usage scenarios. It then triggers the Darshan shutdown routine and measures the cost of each step.

Figure 5 plots the shutdown time with ten samples at each scale for two shared file scenarios: one in which each process simulates accessing a single shared file and one in which each process simulates accessing a total of 1,024 shared files. Both the x and y axes are using a log scale. Except for one outlier at the 24,576 processes, no example took longer than 0.26 seconds to complete even with 98,304 MPI processes. Note that for globally shared files, all data is reduced to the rank 0 process before being written to disk.

TABLE 2
AVERAGE PERCENTAGE OF TIME SPENT IN DARSHAN SHUTDOWN PHASES FOR 98,304 PROCESS EXAMPLES

Scenario	Total Time	Aggregate	Compress	Store
1 shared file	0.17 s	3.3%	0.1%	95.3%
1,024 shared files	0.24 s	23.1%	9.4%	66.9%
1 file-per-proc	12.27 s	0.2%	0.0%	99.8%
1,024 files-per-proc	24.76 s	0.0%	0.1%	99.8%

Figure 6 shows the results of the same experiments in two file-per-process scenarios: each process opens one unique file, and each process opens 1,024 unique files. Unlike the shared file examples from Figure 5, Darshan is unable to reduce the total number of file records because they are not shared. We therefore see that the Darshan shutdown cost grows with the total number of files accessed. The 1,024 unique files example at 98,304 files is simulating an application that opened a total of over 100 million distinct files. This worst-case scenario took an average of 24.7 seconds for Darshan to record the complete results.

We note that Darshan shutdown is a one-time cost at the end of job execution and does not affect performance up to that point. Darshan will also fall back to a compact, less granular instrumentation format that no longer distinguishes between files if more than 1,024 unique files are opened per process, at which point the shutdown overhead would more closely resemble the single shared file example. This 1,024 file limit could be lowered in order to constrain the Darshan overhead further if applications are expected to routinely open such a large number of files.

Table 2 summarizes the time spent in each phase of Darshan shutdown for the 98,304 process examples. The majority of time is generally spent actually writing data. This step includes the time needed to create the output file, write all data to it using collective MPI-IO operations, close the file, and rename it. The 1,024 shared file case diverges from the others in that it spends a larger proportion of time aggregating file records, which is to be expected as it must aggregate all 1,024 file records across all processes prior to writing results. Although aggregation accounts for a significant percentage of shutdown time in this case, it still improves overall performance by drastically reducing the volume of data to be stored.

V. DEPLOYMENT EXPERIENCE

Darshan is deployed at NERSC on Hopper system as a module. The Darshan module was loaded as part of the default environment for all NERSC users starting on November 15, 2012. The initial deployment only instrumented executables compiled using the PGI compiler. On January 7, 2013, Darshan was enabled for GNU and Intel compilers, and finally on February 1, 2013, Darshan was enabled for the Cray compiler as well. Figure 7 shows the percentage active users and number of jobs logged by Darshan. Figure 8 shows the percentage raw compute hours logged by Darshan. By the end of March 2013, the percentage of active users logged reached 60% and the percentage of compute hours logged reached 30%. The rate

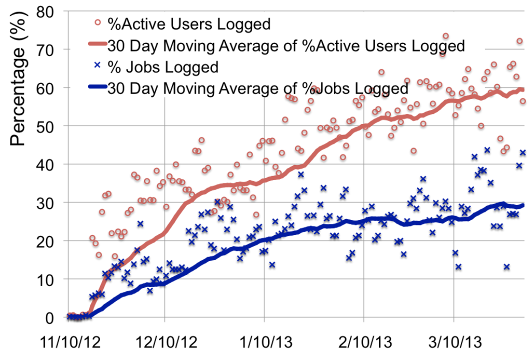


Fig. 7. Percentage of Hopper jobs and active users logged by Darshan. Each circular marker on the plot shows the percentage of active users that have at least one job logged by Darshan on a particular day. An active user is defined as a Hopper user that successfully completes at least one job. Each cross marker on the plot shows what percentage of jobs are logged by Darshan.

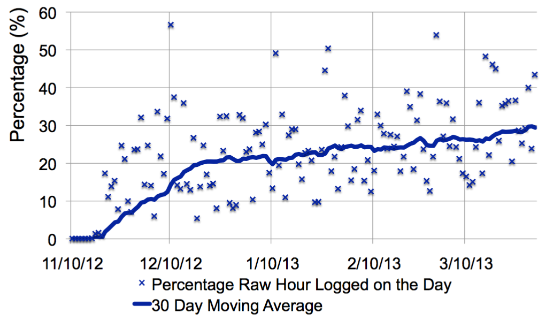


Fig. 8. Percentage of Hopper raw hours logged by Darshan.

of increase in coverage is limited because a large number of NERSC users are running precompiled executables and do not frequently recompile.

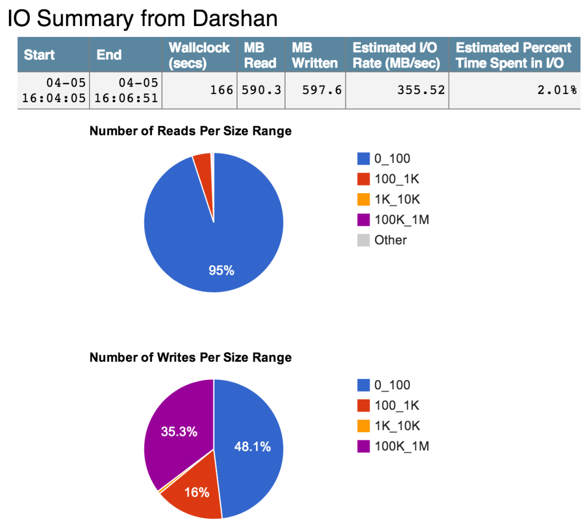
The Darshan logs are stored in a central location on the Hopper Scratch Lustre file system. A cron job monitors the log location and inserts the Darshan aggregate performance data into a database every five minutes. Every morning the log files of the previous day are archived. In order to minimize metadata server contention on the Lustre file system, old logs are backed up and deleted.

NERSC has a user-accessible web portal that displays information about completed jobs, which now includes Darshan statistics when available. The Darshan report for each job shows a subset of characterization data including the total data read and written, estimated I/O rate, estimated percentage of time spent in I/O, and a distribution of read and write transaction sizes. A screenshot of the Darshan summary from the NERSC web portal is shown in Figure 9.

We have also implemented a preliminary automatic filtering mechanism to identify inefficient I/O jobs and notify administrators so that they can proactively help users with I/O problems.

VI. Case study

Although data collected using Darshan can be used for a number of purposes by administrators, researchers, and end-


IO Summary from Darshan

Start	End	Wallclock (secs)	MB Read	MB Written	Estimated I/O Rate (MB/sec)	Estimated Percent Time Spent in I/O
04-05 16:04:05	04-05 16:06:51	166	590.3	597.6	355.52	2.01%

Fig. 9. NERSC user portal showing Darshan statistics for an example job.

users, one of the most compelling potential use cases is to automatically identify underperforming applications and provide guidance on how they might improve their I/O behavior. This is a daunting task, however, as applications vary considerably across scientific domains and many factors (some beyond the user's control) can influence performance.

In this section we explore three example metrics that can be used to help identify applications that exhibit suboptimal I/O behavior. These metrics are computationally simple to calculate and could be generated immediately following the completion of a job for near real-time classification. They are not guaranteed indicators of under-performing applications, but they can be used to speed the process of identification and tuning by identifying likely candidates for further inspection. The example metrics shown here are also not exhaustive; they are proofs of concept to demonstrate the type of automated analysis that could be enabled by using production Darshan characterization. The remainder of this section describes each metric, applies it to a collection of production logs gathered between January 1 and March 13, 2013, and investigates the behavior of an example application that triggered the metric.

A. *Redundant I/O traffic*

One access pattern that may indicate suboptimal use of storage resources is the practice of accessing the same data from the file system multiple times, either by rereading the same data or rewriting the same data. This may be a valid access pattern in some cases (e.g., out-of-core algorithms, data sieving optimizations, and I/O benchmarking), but in other cases it indicates that an application is making inefficient use of shared file system resources. Redundant I/O traffic can often be eliminated through parallel I/O or aggregation strategies.

One category of redundant I/O traffic that can be quickly identified by Darshan log analysis is the scenario in which an application reads a volume of data from a file that exceeds the size of the file. For example, if an application reads

TABLE 3
JOBS IDENTIFIED USING REDUNDANT READ METRIC

Redundant read threshold	> 1 TiB
Total jobs analyzed	261,890
Jobs matching metric	671
Unique users matching metric	37
Largest single-job redundant read volume	547 TiB

16 MiB of data from a 1 MiB file, it indicates that the application is issuing duplicate read operations that could likely be eliminated in order to improve performance. Darshan tracks two per-file access pattern characteristics that can help identify this scenario: the maximum offset that was read from a file and the total number of bytes read from a file. If the latter is greater than the former (especially if no data was written to the file), it indicates the presence of redundant file system traffic.

We can therefore identify the amount of redundant read traffic for a given log using the following equation for all files such that $bytes\_read$ exceeds max_offset_read.

$$\sum_{n=1}^{nfiles} bytes\_read - max\_offset\_read - 1 \qquad (1)$$

Using this equation, we can rank all jobs instrumented with Darshan in terms of their estimated volume of redundant read traffic. Table 3 summarizes the collection of jobs that were found to generate over 1 TiB of redundant read traffic.

A closer analysis of the top example in this set shows that it was a physics application that executed on 6,138 MPI processes for 6.5 hours. This was a read-intensive application: each process spent an average of almost 27 minutes reading an aggregate total of 548.6 TiB of data from a collection of files containing a total of no more than 1.2 TiB of relevant data according to the maximum offset read from each file. This indicates that the application may have generated as much as 457 times as much traffic on the I/O network as was strictly needed to transfer data from storage to compute resources. The bulk of the redundant read traffic arose from multiple processes reading the same file, but interestingly there was a significant amount of duplicate read traffic at a per-process level as well. Each process read an average of 57% more data than was present in the file.

The bulk of the data was accessed by using approximately 73 billion 8 KiB read operations, most of which were sequential. Data was read from files that averaged 1.1 GiB in size, with a maximum size of 8.0 GiB. There is no indication that this application was using an out-of-core algorithm; each files was accessed exclusively with read-only or write-only operations. There is also no indication of data sieving activity; MPI-IO was not used, and the majority of the access sizes were only 8 KiB in size. Although this application likely benefited extensively from file system caching effects (server-side caching of frequently read data, and client-side readahead), it likely did so at the expense of bandwidth and contention on the I/O network.

TABLE 4
JOBS IDENTIFIED USING METADATA RATIO METRIC

Thresholds	meta_time / nprocs > 30 s nprocs ≥ 192 metadata_ratio ≥ 25%
Total jobs analyzed	261,890
Jobs matching metric	252
Unique users matching metric	45
Largest single-job metadata ratio	> 99%

B. Metadata overhead

In previous studies we have identified that applications that spend a high percentage of time performing metadata operations (such as `open()`, `stat()`, `close()`, and `seek()`) often correlate with low aggregate I/O performance [2]. Darshan tracks the cumulative amount of time spent by each process in metadata and I/O function calls at both the POSIX and MPI-IO layer. We can therefore not only report the aggregate amount of time spent performing metadata operations but also estimate the percentage of total I/O time that was spent performing metadata operations as follows (using either MPI-IO or POSIX counters as appropriate for the access method).

$$\frac{\sum_{n=1}^{nfiles} metadata\_time}{\sum_{n=1}^{nfiles} metadata\_time + IO\_time} \qquad (2)$$

As in the other metrics listed here, there may be valid reasons for applications exhibiting high metadata overhead. I/O cost can also be misreported as metadata cost if the file system elects to flush data to storage at `close()` time. In other cases, high levels of metadata overhead indicate that the application is simply using an inefficient number of small files to store its data. Table 4 summarizes the collection of jobs that spent at least 25% of their I/O time performing metadata operations, used at least 192 processes (8 compute nodes), and spent at least an average of 30 seconds per process performing metadata operations.

Analysis of the top example in terms of metadata ratio shows that it was a climate application executing on 40,960 processes for about 229 seconds. The process that spent the most time performing I/O used 103 seconds of that time performing I/O operations. The job created and wrote to a total of 204,803 files (5 per process) averaging 406 KiB in size. Most files were written by using three `fwrite()` system calls: two of which were 4 bytes in size (possibly a header or trailer) and one large write operation. The `stat()` system call was also issued three times per file for an aggregate count of over 614,649 stat calls. The cost of the `close()` operations was likely inflated by flushing cached data at close time. Darshan records the cumulative time for all metadata operations in a single counter rather than tracking `stat()` and `close()` time separately, so there is no way to discern the relative contribution of each call to the metadata overhead. We expect both to be significant, however. Each process spent an average of over 45 seconds to write less than 2 MiB of data across five files. Performance could likely be improved by

TABLE 5
JOBS IDENTIFIED USING SMALL INDEPENDENT SHARED WRITE METRIC

Thresholds	$>$ 100 million small writes 0 collective writes
Total jobs analyzed	261,890
Jobs matching metric	220
Unique users matching metric	11
Largest single-job small write count	5.7 billion

issuing fewer stat calls and merging the output files generated by each process if possible.

We also note that of the 252 jobs identified using this metric, 72 also appear in the list of 671 jobs suspected to produce significant redundant I/O traffic. The metrics are not necessarily mutually exclusive.

C. Small independent writes to shared files

The task of identifying suboptimal write-intensive access patterns is complicated by several factors. I/O performance may fluctuate naturally based on interjob contention or other sources of variance, as observed in Section IV-A. Small access sizes often correlate with poor performance, but this can also be misleading if the application accesses unique files on each process and the file system can perform write-behind caching. MPI-IO collective operations are also capable of aggregating a large number of small write requests into more efficient access patterns at the POSIX level.

To identify write patterns that were unlikely to be cached effectively or benefit from MPI-IO aggregation, we looked for patterns that involved relatively small write operations (smaller than the default Lustre stripe size of 1 MiB), without the use of MPI-IO collective operations, to shared files. Previous work has shown this access pattern to be suboptimal on Lustre because of the overhead associated with poor alignment to stripe boundaries [8]. We found 220 examples of jobs that issued at least 1 million independent writes smaller than 1 MiB to a shared file. A summary of the matching jobs can be found in Table 5.

The job that generated the largest number of independent small writes to a shared file was a turbulence application that executed on 128 processes for 30 minutes. The slowest process in the job spent over 12 minutes of that time performing I/O operations. The application wrote to 7 different globally shared files. One of them was written using 5.6 billion independent writes between 0 and 100 bytes in size, while the other six were written using 2.9 million independent writes between 0 and 100 bytes in size. The application also wrote to a number of unique files using larger access sizes between 100 KiB and 1 MiB with much better results. Each process wrote an average of 245 MiB to the unique files with large access sizes and 280 MiB to the shared files with small access sizes. The former took an average of of 0.55 seconds per process, while the latter took an average of 264.67 seconds per process. This application appears to be a candidate for collective I/O optimizations when writing to shared files.

VII. RELATED WORK

A number of techniques have been used to analyze I/O activity on Cray systems, including both storage and server-side instrumentation. Kim et al. used statistics gathered from enterprise-class storage devices to uncover characteristics such as bandwidth distributions, interarrival time distributions, and correlation between access size and bandwidth on the Spider storage cluster at Oak Ridge National Laboratory [13]. Uselton and Ushizima used server-side data for statistical analysis and correlation to application-level events on the Franklin Cray XT4 system at NERSC [14]. Uselton et al. have also used server-side data collection to investigate I/O variability and the distribution of read and write workloads across multiple file systems on Franklin [11]. They used recurring 64-process IOR measurements over the course of a year to gauge variability for a known application workload.

Event tracing has also been used to analyze application-level behavior on Cray systems. Shende et al. investigated multiple I/O interposition techniques in the Tau framework and demonstrated profiling of the GCRM application at NERSC [15]. Uselton et al. used the IPM-I/O tracing framework to collect large-scale traces from applications such as MADBench and GCRM on Franklin and then applied statistical analysis to convert the events into performance ensembles to isolate performance bottlenecks [16]. Roth presented the IOT event tracing tool and used it to characterize the I/O behavior of the POP and S3D applications on a Cray XT system [17].

Darshan bridges the gap between server-level monitoring and application-level tracing on Cray systems by capturing application workload statistics in a manner that is lightweight enough for full-time production workloads yet still provides direct insight into the behavior of individual applications.

VIII. CONCLUSIONS

In this work we reported our experiences in tuning and adapting Darshan to the Cray XE6 environment, including compiler script integration, avoiding system call overhead, and optimizing collective I/O. By leveraging these optimizations, which are now available in public Darshan releases, Darshan can be enabled on a production Cray system with negligible overhead relative to system variability for common access patterns with at least 12,288 processes. We further demonstrated that the fixed cost of writing characterization results from 98,304 processes ranges from 0.26 seconds for jobs that open a single shared file up to 24.7 seconds for worst-case jobs that open over 100 million unique files. From these experiments we conclude that Darshan is suitable for full-time production deployment on the Cray XE6 platform. This enables a unique ability to transparently instrument production I/O behavior on Cray systems with application-level granularity. We also quantified I/O variability on Hopper for both shared-file and file-per-process workloads with 12,288 processes as part of our study. This is an important consideration for I/O researchers to take into account when measuring I/O performance on large-scale systems.

Darshan was successfully deployed for all users on the Hopper system at NERSC in December of 2012. We used the logs collected on Hopper from January 1 to March 13, 2013, as a case study in exploring metrics for identifying applications that could benefit from additional I/O tuning effort. We demonstrated the use of three example metrics to identify jobs with redundant read patterns, excessive metadata overhead, or suboptimal shared-file write patterns, finding significant exemplars in all three cases.

In future work we intend to use metrics such as those explored in this work to guide improvements in scientific productivity for production applications. We also plan to improve the Darshan tool itself by integrating file system-specific APIs in order to more efficiently gather statistics relevant to I/O characterization when possible. Darshan will continue to be released under an open-source license as a public, production-ready resource for the community.

ACKNOWLEDGMENTS

This research used resources of the National Energy Research Scientific Computing Center and was supported by the Office of Science of the U.S. Department of Energy under Contract Nos. DE-AC02-05CH11231 and DE-AC02-06CH11357 including through the Scientific Discovery through Advanced Computing (SciDAC) Institute for Scalable Data Management, Analysis, and Visualization.

REFERENCES

[1] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, “24/7 characterization of petascale I/O workloads,” in *Proceedings of 2009 Workshop on Interfaces and Architectures for Scientific Data Storage*, September 2009.

[2] 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, p. 8, 2011.

[3] Argonne Leadership Computing Facility, “Darshan data repository.” [Online]. Available: http://www.mcs.anl.gov/research/projects/darshan/data/

[4] B. Behzad, J. Huchette, H. Luu, R. Aydt, S. Byna, M. Chaarawi, Q. Koziol, Prabhat, and Y. Yao, “Auto-tuning of parallel I/O parameters for HDF5 applications,” Poster at the ACM/IEEE SuperComputing Conference (SC’12), November 2012.

[5] P. Carns, K. Harms, R. Latham, and R. Ross, “Performance analysis of Darshan 2.2.3 on the Cray XE6 platform.” Argonne National Laboratory (ANL), Tech. Rep., 2012.

[6] J. Garlick, “Lustre Monitoring Tool.” [Online]. Available: https://github.com/chaos/lmt/wiki

[7] R. Thakur, W. Gropp, and E. Lusk, “Data sieving and collective I/O in ROMIO,” in *The Seventh Symposium on the Frontiers of Massively Parallel Computation, 1999. Frontiers’ 99*. IEEE, 1999, pp. 182–189.

[8] P. Dickens and J. Logan, “Towards a high performance implementation of MPI-IO on the Lustre file system,” in *On the Move to Meaningful Internet Systems: OTM 2008*. Springer, 2008, pp. 870–885.

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

[10] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf, “Managing variability in the IO performance of petascale storage systems,” pp. 1–12, 2010.

[11] A. Uselton, K. Antypas, D. Ushizima, and J. Sukharev, “File system monitoring as a window into user i/o requirements,” in *Proceedings of the 2010 Cray User Group Meeting, Edinburgh, Scotland*, 2010.

[12] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2012, ISBN 3-900051-07-0. [Online]. Available: http://www.R-project.org

[13] Y. Kim, R. Gunasekaran, G. M. Shipman, D. A. Dillow, Z. Zhang, and B. W. Settlemyer, “Workload characterization of a leadership class storage cluster,” in *5th Petascale Data Storage Workshop (PDSW)*. IEEE, 2010, pp. 1–5.

[14] A. Uselton and D. Ushizima, “Poster: I/O workload analysis with server-side data collection,” in *Proceedings of the 2011 companion on High Performance Computing Networking, Storage and Analysis Companion*, ser. SC ’11 Companion. New York: ACM, 2011, pp. 33–34.

[15] S. Shende, A. D. Malony, W. Spear, and K. Schuchardt, “Characterizing I/O performance using the Tau performance system,” in *International Conference on Parallel Proceedings, Parco Exascale Mini-symposium, Ghent, Belgium*, 2011.

[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 *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 2010, pp. 1–11.

[17] P. C. Roth, “Characterizing the I/O behavior of scientific applications on the Cray XT,” in *Proceedings of the 2nd International Workshop on Petascale Data Storage: held in conjunction with Supercomputing’07*. ACM, 2007, pp. 50–55.