Parallel I/O Performance: From Events to Ensembles

Andrew Uselton†, Mark Howison†, Nicholas J. Wright†, David Skinner†,
Noel Keen†, John Shalf†, Karen L. Karavanic⋆, Leonid Oliker†
†*CRD/NERSC, Lawrence Berkeley National Laboratory Berkeley, CA 94720*
⋆*Portland State University, Portland, OR 97207-0751*

***Abstract*—Parallel I/O is fast becoming a bottleneck to the research agendas of many users of extreme scale parallel computers. The principle cause of this is the concurrency explosion of high-end computation, coupled with the complexity of providing parallel file systems that perform reliably at such scales. More than just being a bottleneck, parallel I/O performance at scale is notoriously variable, being influenced by numerous factors inside and outside the application, thus making it extremely difficult to isolate cause and effect for performance events. In this paper, we propose a statistical approach to understanding I/O performance that moves from the analysis of performance events to the exploration of performance ensembles. Using this methodology, we examine two I/O-intensive scientific computations from cosmology and climate science, and demonstrate that our approach can identify application and middleware performance deficiencies — resulting in more than $4\times$ run time improvement for both examined applications.**

I. Introduction

The era of petascale computing is one of unprecedented concurrency. This daunting level of parallelism poses enormous challenges for I/O systems because they must support efficient and scalable data movement between a relatively small number of disks and a large number of distributed memories on compute nodes. The root cause of an application's poor I/O performance may be found in the code itself, in a middleware library it relies upon, in the file system, or even in the configuration of the underlying machine running the application. Worse, there may be unexpected interplay between these possibilities resulting in significant performance deterioration [13]. The performance of individual I/O events can vary by several orders of magnitude from run to run, making bottleneck isolation and optimization extremely challenging. It is therefore critical for the high performance computing (HPC) community to develop performance monitoring tools and methodologies that can help disambiguate the sources of I/O bottlenecks for supercomputing applications.

In this paper, we propose a statistical approach to understanding I/O behavior that transitions from the typical analysis of performance events to the exploration of performance ensembles. A key insight is that although the I/O rate an individual task observes may vary significantly from run to run, the statistical moments and modes of the performance distribution are reproducible. To efficiently collect parallel I/O statistics in a scalable fashion, we have extended an existing performance tool called IPM (Integrated Performance Monitoring) [19] to add I/O operation tracing (IPM-I/O). IPM is a scalable, portable, and lightweight framework for collecting, profiling, and aggregating HPC performance information.

Using this tool, we evaluate the I/O behavior on large-scale Cray XT supercomputers of the Interleaved-Or-Random (IOR) micro-benchmark as well as two I/O-intensive numerical simulations from cosmology and climate modeling. For the MADbench application, which studies the cosmic microwave background, our approach helps isolate a subtle file system middleware problem, that results in performance improvement of $4\times$. Additionally, our exploration into the 10,240-way I/O behavior of the global cloud system resolving model (GCRM) resulted in several successful optimizations of the application and its interaction with the underlying I/O-library, causing a net performance increase of over $4\times$. Overall our work successfully demonstrates that the statistical analysis of ensembles can be used effectively to isolate complex sources of I/O bottlenecks on high-end computational systems.

A. Related Work

There are several performance tools that measure I/O performance of scientific applications, including KOJAK [10], TAU [18], CrayPat [9], Vampir [16] and Jumpshot [8]. All are general purpose tools that include some I/O measurement and analysis capability. From the variety of tools available there is a wide range in the scope of information collected, performance overhead, and impact on the application being studied. We chose to use IPM-I/O for this study because of its focus upon recording a limited set of metrics in a lightweight manner.

A number of published studies investigate I/O performance on high end systems, as we do in this paper. One investigation [12] characterizes a large-

scale Lustre installation relating poor performance to default striping parameters. Another study of high-end file system performance [20] evaluated the I/O requirements and performance of several applications over 18 months. The often unexpected performance shifts as applications and systems changed over time are a strong argument for the use of scalable, application-centric I/O performance tools and methodologies, as presented in this paper.

Interpreting performance information using statistical techniques has been the subject of several previous works. For example, Ahn and Vetter used a variety of multivariate statistical techniques to analyse performance counter data [5]. This approach is similar in spirit to ours, in that it attempts to combine large amounts of performance information into a more compact representation; however, it does not focus on the specific challenges of understanding large-scale I/O behavior characteristics.

II. Platforms and Tracing Tools

In this section we briefly define the features of our experimental platforms and the IPM-I/O trace tool.

A. Architectural Platforms

Most of the experiments conducted for this study used Franklin, the 9660 node Cray XT4 supercomputer located at Lawrence Berkeley National Laboratory (LBNL). Each XT4 node contains a quad-core 2.1 GHz AMD Opteron processor, which is tightly integrated to the XT4 interconnect via a Cray SeaStar-2 ASIC through a 6.4 GB/s bidirectional HyperTransport interface. All the SeaStar routing chips are interconnected in a 3D torus topology, where each node has a direct link to its six nearest neighbors. Franklin employs the Lustre parallel file system as its temporary file systems `scratch` and `scratch2` each with 24 Object Storage Servers (OSSs) and with 2 Object Storage Targets (OSTs) on each OSS.

Additionally, several experiments were conducted on Jaguar, the Cray XT4/XT5 system located at Oak Ridge National Laboratory (ORNL), which also uses Lustre. The combined Jaguar system has 7832 nodes in the XT4 portion and another 37,544 nodes in the XT5 portion. The results reported in Section IV use the XT4 portion with 72 OSSs hosting 2 OSTs each for a total of 144 OSTs.

B. IPM-I/O Tracing

IPM has previously [19], [21] been used to understand computation, communication, and scaling behavior of parallel codes. These studies focused on performance limitations related to the compute node resources (caches, memory, CPU), messaging and switch contention, and algorithmic limitations. I/O brings with it a new set of shared resources in which contention and performance variability may occur. Metadata locking, RAID subsystems, file striping, and other factors compound the complexity of understanding measured wall clock times for I/O. While contention for node and switch hardware resources do affect, for example, MPI performance, the impact of contention upon parallel I/O at scale is much more prevalent and significant, mostly because there are (generally) relatively few I/O resources compared to computational ones.

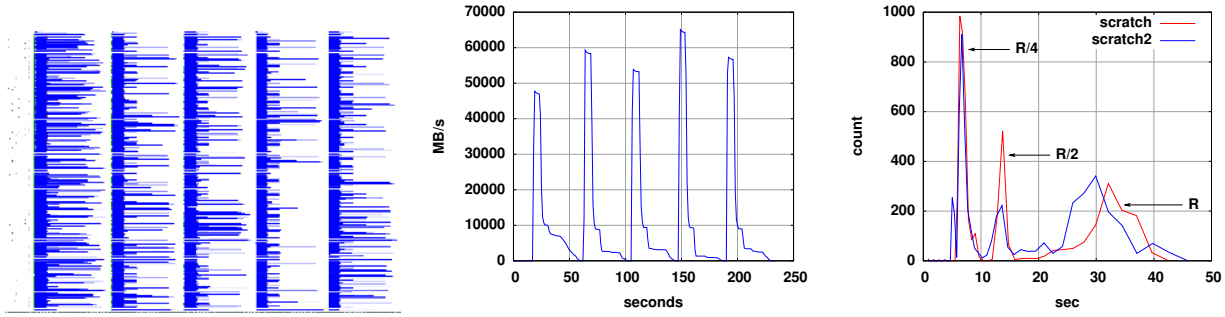
The work reported here employs newly developed I/O functionality for IPM to generate trace data in a lightweight, portable, and scalable manner. IPM-I/O works by intercepting an application's POSIX I/O calls into the `libc` library. To use IPM-I/O an application is linked against the IPM-I/O library using the `-wrap` functionality of the GNU linker, which provides a mechanism for intercepting any library call. In this case it redirects POSIX I/O calls to IPM-I/O. During the application run IPM-I/O collects timestamped trace entries containing the `libc` call, its arguments, and its duration. A look-up table of open file descriptors allows IPM-I/O to associate events interacting with the same file. By default IPM-I/O emits the entire trace. This approach has proved to be scalable for I/O tracing applications running with up to 10K MPI tasks without any significant slowdown being observed.

III. Performance Ensembles

Most parallel I/O in High Performance Computing (HPC) is distinct from the random access seen in transaction processing and database environments [11]. From profiling workloads at supercomputing centers [17] we have observed that HPC I/O in this environment frequently involves large-scale data movement, such as check-pointing the state of the running application. Furthermore, reviewing user requirements documents confirms this observation, as do complaints from HPC users.

In a production supercomputing environment, it is common that observed details of I/O performance change from one application run to the next. Factors affecting performance include the load from other jobs on the HPC system, task layout, and multiple potential levels of contention, among numerous others. Our goal is to determine robust ways of examining I/O performance that are stable under the changing conditions from one run to the next. To this end we examine the distribution of individual I/O rates observed during a test, and study the statistical properties of the distributions.

To present an example of this approach, we first examine results using the Interleaved-Or-Random (IOR)



(a) I/O trace diagram **(b) Aggregate I/O rate** **(c) I/O time distribution**

Figure 1: IOR 512 MB transfers using 1024 processors. a) The trace diagram: the y-axis represents the tasks (1 - 1024) and the x-axis represents time in seconds; blue indicates time spent in write() and white space indicates all other time. This diagram shows 5 phases of I/O. b) The aggregate data rate over all tasks (y-axis) is plotted versus wall clock time (x-axis). c) The two distributions each count (y-axis) events for a given amount of time (x-axis), on different Franklin parallel file systems, `scratch` **and** `scratch2` **($R = \frac{512MB}{16MB/s}$).**

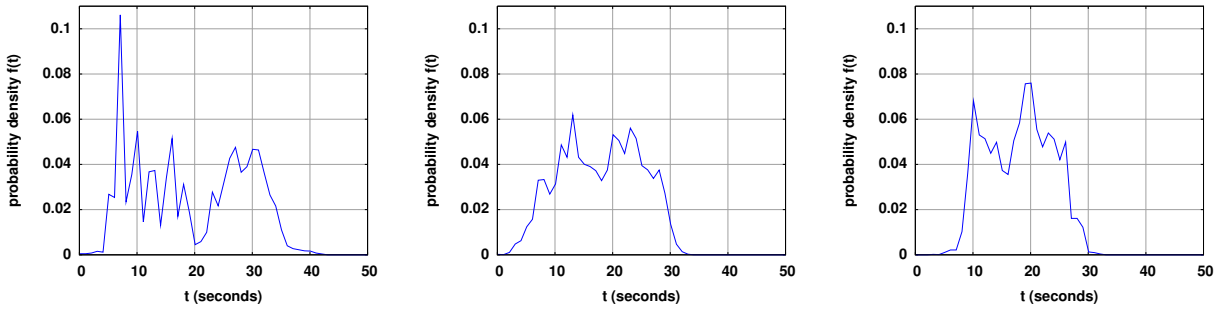
[14] code. IOR is a parametrized benchmark that performs I/O operations for a defined file size, transaction size, concurrency, I/O-interface, etc. For our experiments, shown in Figure 1, IOR has been configured to run with 1024 tasks on 256 nodes of Franklin. Each task writes 512 MB to a unique offset within a shared file, and does so in a single `write()` call, followed by a barrier. This is then repeated five times. The IOR binary has been augmented with the IPM-I/O library to capture I/O events. In this context we refer to a particular choice of test parameters as an *experiment* and a specific instance of running that experiment simply as a *run*.

Figure 1(a) depicts the I/O traces for five runs of the experiment, all in a single job. Each task's time history is represented with a separate horizontal line, with task 0 at the top and task 1023 at the bottom. The *x-axis* is wall-clock time and each trace proceeds from left to right showing the I/O pattern from the beginning to the end of the test. Each bar corresponds to the *write* (blue) of 512 MB, and its length gives the duration of the I/O. White space represents non-I/O activity, which is a barrier wait in these experiments. Since all of the `write` calls transfer the same amount, short bars represent fast I/O and longer bars slower I/O performance.

The trace in Figure 1(a) shows two phenomena common to HPC I/O at scale. The first is that the synchronous nature of many applications leads to vertically banded intervals during which parallel I/O occurs, i.e., the I/O happens in synchronous phases. As a consequence the task that arrives last at the barrier will define the performance of the application for that phase. Thus a small number of events, or even a single event, can define the performance of an application. The second is that there is great variability in the performance of individual (theoretically identical) I/O events. This variability appears to be random in the sense that a given individual MPI task is not consistently slow or fast. Figure 1(b) shows the instantaneous data rate, across the 1024 tasks taken together, over the life of the job. There does seem to be a consistent initial high plateau around 60 GB/s followed by another brief plateau around 10 GB/s and a final long tail that is slower still.

In this situation a statistical representation of the I/O events provides a clearer picture of the overall performance, as shown in Figure 1(c). This is a histogram of the distribution of completion times for individual I/O events shown in Figure 1(a), which is from the `scratch` file system on Franklin. Figure 1(c) also shows the distribution from `scratch2`. We note that the statistical representations are almost identical, but the second run of the same experiment on the different file system produces a trace very different it its specific details, albeit with a similar overall run time.

Observe that each histogram has three prominent peaks corresponding to three distinct modes of behavior. The aggregate available data rate from either of the two file systems is limited to about 18 GB/s by the network infrastructure, and observed peak performance is somewhat short of that. Suppose each of the 1024 tasks got a *fair share* of this aggregate data rate. For a task to move 512 MB in 30 to 32 seconds, as the peak labeled "R" indicates, means that task saw about 16 or 17 MB/s — close to the fair share of the peak available data rate. Note that the other two strong peaks are at the



(a) two 256 MB transfers **(b) four 128 MB buffers** **(c) eight 64 MB transfers**

Figure 2: IOR 512 MB transfer using 1024 processors where: a) 512 MB written via two 256MB `write()` calls. b) Four calls (128MB). c) Eight calls (64MB). Note that the distributions become progressively narrower and more Gaussian.

second and fourth harmonic for this rate, which implies that one task on the node (or two) took all the available I/O resources until it was done, with the other tasks waiting until it was complete. This implies a particular order to the processing in the Lustre parallel file system. Note further that these three peaks do not correspond to the three plateaus from Figure 1(b). Those modes reflect filling local system buffers and then having the off-node communication throttle back the data rate.

Overall, the modes in Figure 1(c) give a much more precise characterization of the I/O behavior, thus increasing the potential for appropriate diagnosis and remediation (where appropriate). We conclude that while the performance characteristics of individual I/O events can behave erratically, the modes by which they occur are stable. It is this insight that will allow us to see past the seemingly random individual I/O performance measurements to address potential bottlenecks. This transition from mechanistic analysis of isolated systems of events to the analysis of ensembles resembles the successful strategy of statistical physics whereby large numbers of interacting systems can be described by the properties of their ensemble distributions such as moments, splittings and line-widths.

A. Statistical Analysis

In the upcoming discussions on the statistics of observed I/O times we allude to two commonplace observations about statistical ensembles. The first is *Order Statistics* — in particular, the N^{th} order statistic for a sequence of N observations is the largest value in the ensemble, and its distribution $f_N(t)$ is given by:

$$f_N(t) = NF(t)^{N-1} f(t) \quad (1)$$

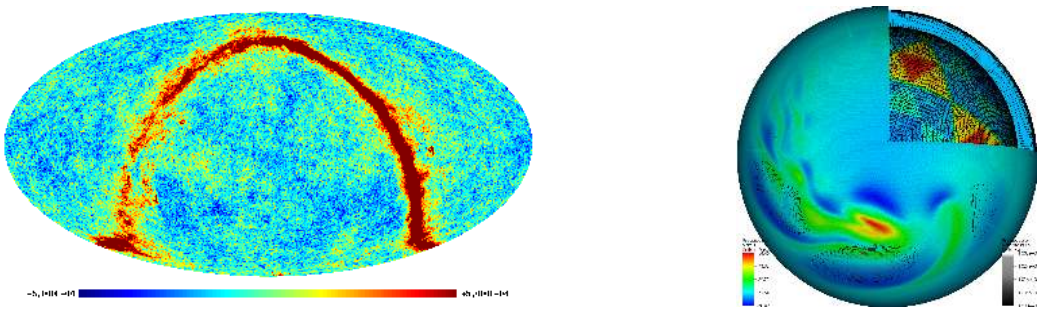
where $f(t)$ is the probability density function for the I/O time for one observation, and $F(t)$ is the corresponding cumulative probability distribution. $f_N(t)$ gives the distribution for the *longest* observation given the underlying distribution.

As N increases the expression $F(t)^{N-1}$ quickly converges to a step function picking out a point in the right-hand tail of the distribution $f(t)$. The distributions in Figure 1(c), when normalized, give an approximation to the probability density function $f(t)$ from Equation 1, and the cumulative probability distribution $F(t)$ is the integral of $f(t)$ (see Figure 5(a)).

The second observation concerns an application of the Law of Large Numbers. Let $T_i | i \in \{1 \ldots k\}$ be the time to completion for a sequence of k I/O operations governed by independent identical distributions with average μ, and let the completion time t_k for the sequence be the sum of the individual observed I/O times $t_k = \sum_{i=1}^{k} T_i$. The expected value of t_k will converge to $k\mu$ as k increases. In other words, the more samples one takes from a distribution, the closer the sample average will be to the average of the underlying distribution.

Figure 2 shows three probability density functions for a sequence of experiments comparable to that illustrated in Figure 1. These are the distributions of t_k values measured over all the MPI tasks for three IOR experiments in which the 512 MB is sent to the file system in k = 2, 4, and 8 successive `write()` calls (using 256, 128, 64MB respectively) — with no barrier until all 512 MB has been written. The run time for an experiment, and therefore the reported data rate, is determined by the slowest I/O operation amongst all the tasks. In each case the slowest is the N^{th} order statistic mentioned in Equation 1 for the corresponding t_k.

As the value of k increases the slowest running task becomes a little faster. In the case of a single 512 MB write, the run time is approximately 45 seconds (see Figure 1(c)) and the reported data rates for the 512 MB



(a) MADCAP (b) GCRM

Figure 3: (a) Visualization of high resolution cosmic microwave background sky map, used by MADCAP to compute the angular power spectrum [6] (b) A pseudocolor plot of a wind velocity variable from a GCRM data set displayed using the VisIt visualization tool.

experiments is around 11,610 MB/s. The reported data rate for the 256 MB experiments is 12,016 MB/s, or about 3% faster. More and smaller transfers continue the trend with 128 MB experiments getting 13,446 MB/s and 64 MB experiments achieving 13,486 MB/s — a 16% speedup.

Since the underlying I/O activity in each of these experiments is the same, it is reasonable to think that dividing the I/O up into multiple `write()` calls would have little on no effect on the overall performance. In fact one might even expect a small penalty for the extra system call processing. However, this is not the case. The worse case behavior improves as k increases because the distributions are getting narrower. That in turn is a consequence of the Law of Large Numbers. In other words, the more opportunities a task has to sample, the more likely it is to have average performance.

We now explore two scientific computations, MADbench and GCRM, and show how our statistical methodology can be used to identify bottlenecks and increase I/O performance.

IV. MADbench I/O Analysis

We apply the forgoing insights to the analysis of two important HPC applications, starting with the Microwave Anisotropy Data-set Computational Analysis Package (MADCAP). MADbench is the second generation of a HPC benchmarking tool [7], [15] that is derived from MADCAP which is an application that focuses on the analysis of massive cosmic microwave background (CMB) data sets. The CMB is the earliest possible image of the Universe, as it was only 400,000 years after the Big Bang. Extremely tiny variations in the CMB temperature and polarization encode a wealth of information about the nature of the universe, and a major effort continues to determine precisely their statistical properties. The challenge here is twofold: first the anisotropies are extraordinarily faint, at the milli- and micro-K level on a 3K background; and second it is necessary to measure their power on all angular scales, from the all-sky to the arc minute. As illustrated in Figure 3(a), obtaining sufficient signal-to-noise at high enough resolution requires the gathering — and then the analysis — of extremely large data sets. Therefore CMB analysis is often extremely I/O intensive.

MADbench is a lightweight benchmark derived from MADCAP that abstracts the I/O, communication, and computation characteristics to facilitate straightforward performance tuning. It is an out-of-core solver that has three phases of computation. During the first phase it generates a series of matrices and writes them to disk one by one. In the middle phase, MADbench reads each matrix back in, multiplies it by an inverse correlation matrix and writes the result back out. Finally, MADbench reads the result matrices and calculates a trace of their product. In the experiments reported here all computation and communication has been effectively turned off, so we can focus exclusively on the I/O component.

Each transfer of a matrix to or from the file system consists of a single large write or read, which is about 300 MB per MPI task in the experiments reported here. The write or read is performed using an MPI-IO call (*MPI_File_write* and *MPI_File_read*). Each task manages and computes values for its portion of a sequence of such matrices — eight of them in these experiments — and performs I/O to an exclusive region within a shared file. All matrices for a task are sequentially ordered in a contiguous file region (modulo an alignment parameter, which is 1 MB in these experiments). Overall the I/O pattern from each MPI task looks like this: 8× (write 300MB), 8× (seek, read 300MB, seek, write 300MB), 8× (read 300MB). This I/O pattern is atypical in that it is sensitive to both read and write I/O rates, most of the available memory

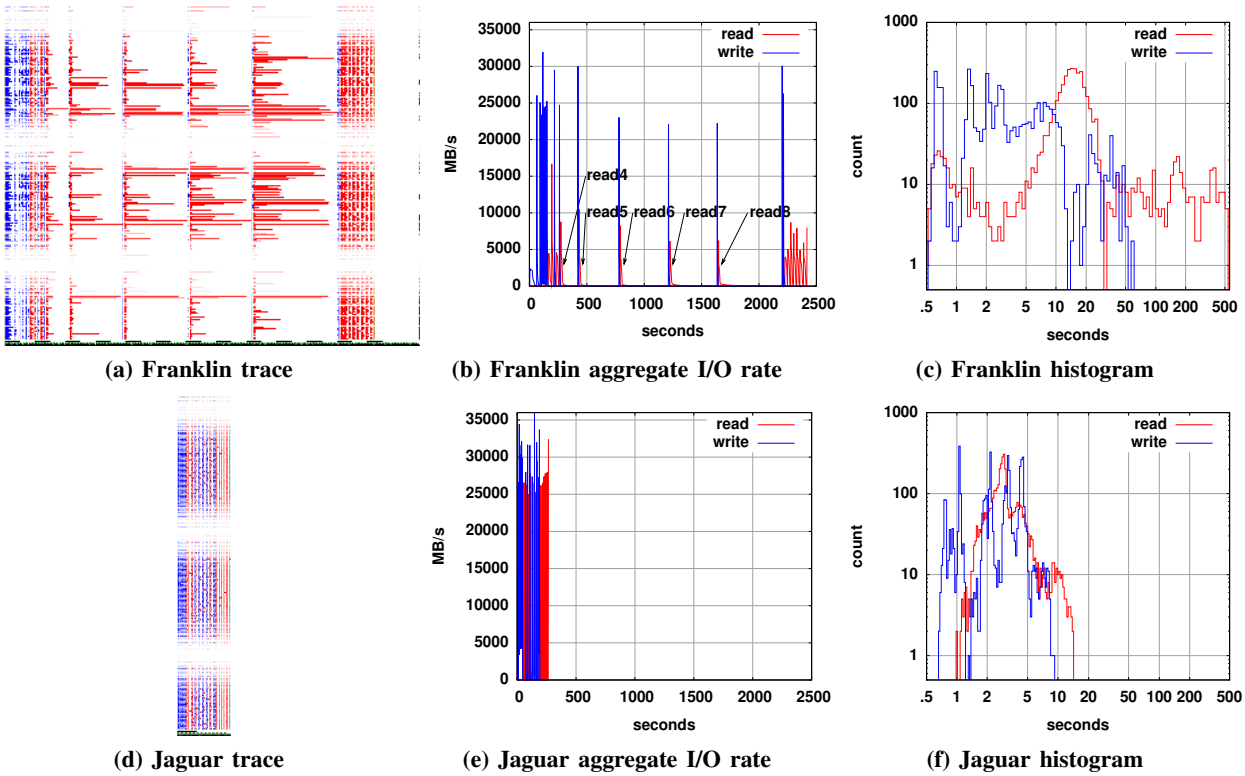


Figure 4: MADbench 256-task experiment on Franklin (2200 seconds) and Jaguar (275 seconds), showing the trace data, aggregate I/O rate, and I/O histogram for each platform. Franklin's slow reads are seen in the broad right shoulder of the read rate distribution in (c).

is already in use, circumventing file system caching efficiencies, and the pattern of *seek, read, seek, write* in the middle phase of the computation is not a streaming I/O pattern.

Previous work [7] shows that MADbench exhibits significantly different performance characteristics under various choices of operating mode and hardware platforms.

A. Trace-Based Analysis

Figure 4 depicts the I/O traces for two MADbench single-file experiments at 256 tasks* on Franklin and Jaguar XT4 systems. The I/O traces in Figures 4(a) and 4(d) were generated via IPM-I/O as described in Sections II-B and III. In these figures each bar corresponds to the *write* (blue) or *read* (red) of a 300 MB matrix, and its length gives the duration of the I/O†. White space represents a barrier wait. Since all of the matrices are the same size, short bars represent fast I/O and longer bars slower I/O, and the overall performance is again dominated by the slowest individual performers.

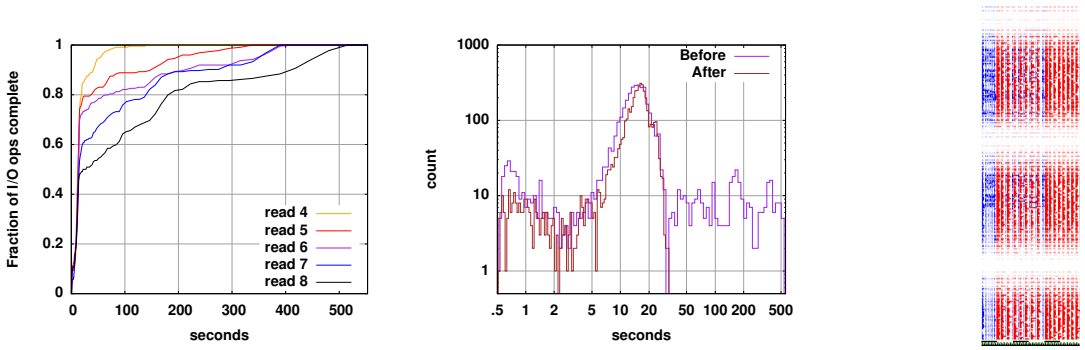
The I/O hardware and software infrastructure is different enough between the two systems that a significant difference both in I/O pattern and aggregate time to run the application is apparent. Jaguar, (Figure 4(d)), shows only modest variability in I/O rate from one task to the next, whereas Franklin (Figure 4(a)) shows a much larger variation in I/O performance from one task to the next. Also the reads in the sequence (seek-read-seek-write) in the middle third of the computation on Franklin are sometimes slow, whereas those at the end, where the sequence is simply eight reads one after the other, show little variability. (We note here that this is not simply a quirk of a single run, it occurs at multiple concurrencies and is reproducible.)

B. Data-Rate Analysis

The slow reading tasks in Franklin's I/O trace, Figure 4(a), that cause the long delays stand out sharply, and those diagrams give an intuitive view of the behavior of the application. The difference between the performance on the two machines is also illustrated by comparing Figures 4(b) and 4(e) which show the

*Traces at other concurrencies show qualitatively similar behavior.

†The attentive reader will note that the middle phase actually begins with two reads and ends with two writes. See [7] for details.



(a) Read rate deterioration **(b) Read performance before and after middleware update** **(c) Franklin trace after update**

Figure 5: MADbench 256-task Franklin experiment before and after middleware update a) The fraction of I/Os completed versus time deteriorates from read 4 to read 8, leading directly to the discovery of a subtle system software (Lustre) bug. b) Read histogram before and after bug correction c) Trace file after update showing removal of catastrophic delays.

instantaneous read and write rates on the two machines. On Franklin the overall duration of each read increases from the fourth read (read4) to the eighth read (read8) and each of these reads has a long tail that continues until the next write phase.

Figures 4(c) and 4(f) show the histograms for Franklin and Jaguar respectively. In this case the histograms are presented as *log-log* plots so that the different modes, especially the slowest modes, stand out. The histograms for `write()` calls on Franklin and Jaguar in Figure 4 are similar and both show four strong peaks on the left with less prominent features trailing to right.

Note that the two write (blue) distributions in Figures 4(c) and 4(f) display similar performance characteristics, while the read (red) distributions show a markedly different pattern from each other. For the Franklin experiment the slowest `read()` calls vary from 30 to 500 seconds. It is these expensive reads that stand out as anomalies in Figures 4(a) and 4(b). The reads in Figure 4(c) centered around the peak at 15 seconds do not show the usual rounded-peak shape expected for a mode with some variability. Instead, this is either several poorly resolved peaks next to each other, or some broad and flat mode unlike the rest.

C. Performance Resolution

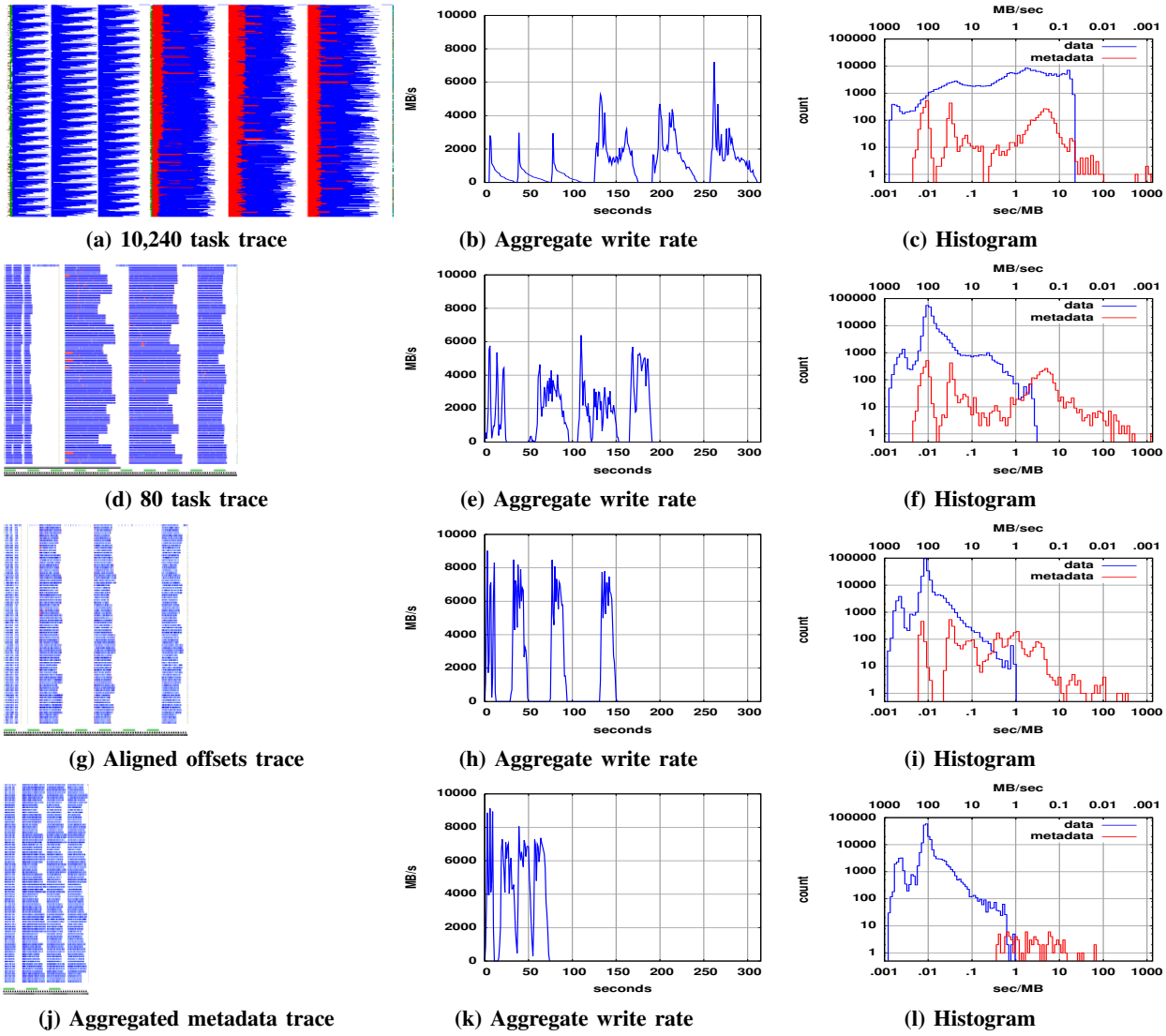
The slow reads on Franklin in Figures 4(a) and 4(c) all occur in the fourth through eighth reads, as shown in Figure 4(b). In Figure 5(a) those reads are presented separately. Figure 5(a) presents the cumulative probability distribution $F_p|p \in \{4,5,6,7,8\}$ for the reads in these phases. That is, each curve in Figure 5(a) gives the progress of I/O during the phase versus time. Not only are the slow reads confined to reads 4 through 8, but they get progressively worse. These two insights lead directly to determining the source of the bottleneck.

The MADbench I/O pattern aligns each I/O operation to a 1 MB boundary, and that produces a small gap between the end of each I/O region and the next. This strided pattern is one that the Lustre parallel file system recognizes and takes into account. In particular, the strided I/O pattern is recognized by Lustre on its third appearance. Subsequent reads that match the stride (the fourth and after) get a larger read-ahead window. In the phase where reads alternate with writes the client-side system buffers were all full, and Lustre issues one page (4 kB) reads due to a lack of system memory resources. This large number of small reads lead to the expensive delays. The later reads did not suffer this effect because system memory was not being filled with interleaved writes.

Due to our investigation, a patch was created for the Lustre file system that avoids the erroneous window-size calculation and that patch was installed onto the Franklin system. The patch removed strided read-ahead detection entirely and the associated expensive delays. This improved the overall performance by more than 4.2×. In Figure 5(b) the distribution for reads after the Lustre patch is applied is superimposed on the read distribution from Figure 4(c). It is clear that the problem has been resolved, as also seen in the trace file of Figure 5(c), where the job run time has been reduced from 2200 seconds to 520, and the trace is comparable to that obtained from Jaguar.

V. GCRM I/O Analysis

The Global Cloud Resolving Model (GCRM, Figure 3(b)), is a climate simulation developed by a



(a) 10,240 task trace (b) Aggregate write rate (c) Histogram

(d) 80 task trace (e) Aggregate write rate (f) Histogram

(g) Aligned offsets trace (h) Aggregate write rate (i) Histogram

(j) Aggregated metadata trace (k) Aggregate write rate (l) Histogram

Figure 6: GCRM using 10,240 tasks writing to a shared file, showing trace graph, aggregate I/O write rate, and histogram distribution for baseline configuration and three progressive optimizations: (a–c) Baseline configuration. (d–f) Data written by only 80 tasks. (f–h) Writes padded and aligned to 1MB boundaries. (j–l) Metadata writes are aggregated into a few large writes.

team of scientists at Colorado State University led by David Randall [2]. It runs at resolutions fine enough to accurately simulate cloud formation and dynamics and, in particular, resolves cirrus clouds, which strongly affect weather patterns, at finer than 4km resolutions. Underlying the GCRM simulation is a geodesic-grid data structure containing nearly 10 billion total grid cells, an unprecedented scale that challenges existing I/O strategies. Researchers at Pacific Northwest National Lab [1] and LBNL [3] have developed a data model, I/O library, and visualization pipeline for these geodesic grids, as well as a GCRM I/O kernel for tuning I/O performance.

Initially, the I/O library was able to achieve only around 1 GB/s, a fraction of the available write rate on Franklin. In order for I/O to consume less than 5% of the total GCRM simulation run time at 4 km resolution, the GCRM I/O library must sustain at least 2GB/s, and preferably more to facilitate scaling to finer resolutions. Therefore we employed the diagnostic tools and methods discussed in earlier sections to investigate and improve performance. Based on this analysis, we worked with the Hierarchical Data Format (HDF) Group to optimize the GCRM I/O kernel.

Our baseline configuration uses 10,240 MPI tasks, each writing the same amount of data, representing different GCRM variable types. This lead to an I/O pattern with three writes of a single 1.6 MB record, each followed by a barrier, then three writes of six 1.6 MB records, followed by another barrier. All the data was written to a single shared file using H5Part [4], a simple data scheme and veneer API built on top of the HDF5 library.

Figures 6(a)–6(c) present the trace graph, write rates and histogram for the baseline. Figure 6(a) shows the limited value of a trace graph at this scale; resolving in detail each of the 10,240 stacked horizontal lines is extremely difficult. In particular, it is not apparent that most of the graph is actually white space. HDF5 metadata operations account for the read activity seen in red in the trace graph.

Figure 6(b) shows that the baseline achieved only a fraction of Franklin's available 16 GB/s aggregate write rate. The peak rate is barely half that, and most of the run time was spent at rates of less than 2 GB/s. Once again, the I/O pattern is governed by the worst-case behavior.

The statistical view is essential to understanding the behavior of the code. In the histogram (Figure 6(c)), we plot separate distributions for two buffer sizes, one corresponding to GCRM records (1.6 MB, blue) and the other to HDF5 metadata (<3 KB, red). Unlike the experiments in the previous sections, there are multiple transfer sizes plotted in the histograms, so we normalize the histograms to present MB/sec along the top and sec/MB along the bottom. Faster writes still appear on the left and slower ones on the right. Each of the 10,240 tasks should ideally access a fair share of approximately 1.6 MB/s, given the available 16GB/s aggregate rate. Unfortunately, the baseline exhibits a distribution of per-task data rates with broad peaks well below 1 MB/s and extending to around 0.5 MB/s. The sustained write rate over the entire run time was only about 1 GB/s.

The first optimization follows directly from the insight gained in the experiments of Figure 2. Because each task is executing a small number of writes, we can benefit from a "collective buffering" scheme (similar to that of MPI-IO) in which the data is aggregated from all tasks to a smaller subset of I/O tasks using MPI communication (stage one) then written to disk using only the I/O tasks (stage two). In previous IOR tests on Franklin (not reported here), we observed that as few as 80 tasks can saturate the I/O subsystem. Therefore, we tested a collective buffering scheme (stage two only) by running the I/O kernel with 80 tasks, each with $\frac{10240}{80}$ = 128× as many write calls. The number, size, and alignment of the write calls remained unchanged from the baseline, as did the total amount of data written. Performance was improved due to the Law of Large Numbers advantage described in the IOR experiments in Section III.

The results of this optimization are shown in Figures 6(d)–6(f); the total run time dropped from 310 seconds to 190 seconds, a 1.6× speedup. Figure 6(e) shows that the peak data rate did not improve, but the overall rate is more consistent with less fall off. The peak of the per-task rate distribution (Figure 6(f)) is 100 MB/s, which corresponds to an aggregate 8 GB/s for the 80 tasks. The worst case per-task rate has improved: the 128 records that each task transfers prior to the barrier are more likely to average out in performance.

In addition to employing the Law of Large Numbers advantage, this optimization also reduced the number of tasks communicating with the 48 I/O servers from 10,240 to 80, which likely reduces contention and improves I/O server queue depths and service times. However, even with the optimization, the I/O kernel obtained a peak data rate of only 5 GB/s and a sustained write rate of 1.8 GB/s. Thus, we continued our investigation to identify additional opportunities for optimization.

In previous IOR experiments on Franklin, we established that the Lustre file system prefers aligned offsets when writing to a shared file. The Lustre client transfers data to the I/O servers in 1 MB stripes, yet an examination of our trace data (not shown) revealed that the GCRM records were not aligned with these stripes. Using HDF5 library calls, we padded and aligned these

writes to 1MB boundaries. The results are shown in Figures 6(g)–6(i) and show a run time of 150 seconds, less than half that of the baseline. Figure 6(h) shows that the peak write rate has improved and the "bulge" in the Figure 6(f) distribution between 1MB/s and 0.1MB/s has disappeared, leaving the distribution more closely centered around its peak. Similarly, the worst-case per-task rate now lies at 1MB/s rather than 0.5 MB/s. Also, the metadata operations benefited somewhat from alignment with a peak now around 1 MB/s (Figure 6(i)). From Figure 6(g), it is clear that the total run time was dominated by the serialized metadata operations on task 0.

Our final optimization aggregates the metadata writes from many $<$3KB writes into a single 1 MB write that is deferred until file close, rather than at the end of each run. The results of this optimization are shown in Figures 6(j)–6(l). The large gaps caused by serialized writing on task 0 have disappeared and, the total run time has decreased to 75 seconds. This is a $< 4\times$ improvement over the baseline.

VI. Conclusions

With the exponential growth of high-fidelity sensor (ex. MADbench) and simulated (ex. GCRM) data, the scientific community is increasingly reliant on ultra-scale HPC resources to handle its data analysis requirements. To use such extreme computing power effectively, the I/O components must be designed in a balanced fashion, as any bottleneck will quickly render the platform intolerably inefficient. However, identifying the root cause of I/O performance deficiencies is an increasingly challenging task, as the source of degradation may be found in the application code, middleware library, file system, underlying architecture — or some combination thereof. To address this concern, we have developed a statistical approach for understanding I/O performance that shifts the analysis from the examination of individual performance events to the study of performance ensembles.

To collect trace data in a production environment, we extended I/O functionality to the IPM profiling tool. Results on large-scale HPC systems, demonstrated that IPM-I/O allowed lightweight, portable, and scalable tracing — effectively collecting I/O statistics for our largest 10,240-way simulation.

Statistical analysis of trace data produced by IPM-I/O shows that the modes and moments revealed by the distribution of I/O times can contribute directly to understanding an application's I/O behavior and potential bottlenecks. An examination of the I/O performance statistics for the IOR benchmark revealed an interesting and surprising I/O boost due to taking advantage of the Law of Large Numbers.

Next, we examined the MADbench cosmology application, which suffered anomalous performance behavior on the Franklin XT4 platform. Using IPM-I/O data collection and our performance histogram methodology allowed us to identify a Lustre file system bug, which caused an erroneous read-ahead window. The ability to isolate this subtle I/O interaction between the application and middleware layer highlights the efficacy of our ensemble approach — and resulted in a 4.2$\times$ MADbench speedup once the appropriate Lustre patch was installed.

Finally, we explored the I/O behavior of a large-scale 10,240-way GCRM climate modeling code. Through our statistical I/O performance analysis, we discovered a series of application-level optimizations that dramatically reduced the overall run time from 310 to 75 seconds, an improvement of over 4$\times$.

The three cases described in this paper illustrate the power of our statistics based approach. We fully expect that in future as the number of components in HPC systems increases such approaches will become essential, so that performance measurements can still be tractably recorded and analysed. In fact, the reproducible nature of our performance ensembles suggests that in most cases it may not even be necessary to store a majority of the performance data, just enough to define the distribution.

Future work will build this statistical approach directly into IPM-I/O, thus moving the data captures from an I/O tracing paradigm to an I/O profiling paradigm. This transition promises to improve the scalability of our method in precisely the same way that program counter profiling is more scalable than execution tracing. With the ability to recognize modes and moments of the performance distribution, the IPM-I/O framework will be expanded to detect an application's I/O patterns; thus providing key information to the underlying file system that can be leveraged for improving I/O behavior.

VII. Acknowledgments

We would like to thank Quincey Koziol and John Mainzer of the HDF Group for their assistance with tuning the HDF5 library for the GCRM I/O pattern. We would like to thank Kitrick Sheets of Cray for his support in testing the file system modifications in Section IV. Portions of this work were completed while Dr. Karavanic was at: Performance Modeling and Characterization (PMaC) Laboratory, San Diego Supercomputer Center, La Jolla, CA 92092-0505. This work was supported by the ASCR Office in the DOE Office of Science under contract number DE-AC02-05CH11231; by the Director, Office of Advanced Scientific Computing Research, Office of Science, of the U.S. Department

of Energy under Contract No. DE-AC02-05CH11231 through the Scientific Discovery through Advanced Computing (SciDAC) program's Visualization and Analytics Center for Enabling Technologies (VACET); and by NSF contracts CNS-0325873 and OCI-0721397. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357, resources of the National Energy Research Scientific Computing Center, under Contract No. DE-AC02-05CH11231, and resources of the National Center for Computational Sciences, under contract No. DE-AC05-00OR22725.

REFERENCES

[1] Community Access to Global Cloud Resolving Model and Data. http://climate.pnl.gov/.

[2] Design and Testing of a Global Cloud Resolving Model. http://kiwi.atmos.colostate.edu/gcrm/.

[3] Global Cloud Resolving Model Simulations. http://vis.lbl.gov/Vignettes/Incite19/.

[4] H5Part: A Portable High Performance Parallel Data Interface to HDF5. http://vis.lbl.gov/Research/AcceleratorSAPP/.

[5] D. H. Ahn and J. S. Vetter. Scalable analysis techniques for microprocessor performance counter metrics. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–16, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[6] J. Borrill. MADCAP: The Microwave Anisotropy Dataset Computational Analysis Package. In *5th European SGI/Cray MPP Workshop*, Bologna, Italy, 1999.

[7] J. Borrill, L. Oliker, J. Shalf, H. Shan, and A. Uselton. HPC global file system performance analysis using a scientific-application derived benchmark. *Parallel Computing*, In Press, Accepted Manuscript:–, 2009.

[8] A. Chan, W. Gropp, and E. Lusk. An efficient format for nearly constant-time access to arbitrary time intervals in large trace files. In *Scientific Programming*, volume 16, pages 155–165, 2008.

[9] L. DeRose, B. Homer, and D. Johnson. Detecting application load imbalance on high end massively parallel systems. In *Proceedings, EuroPar 2007, LNCS 4641, pp. 150-159*, 2007.

[10] M. Geimer, F. Wolf, B. J. Wylie, E. brahm, D. Becker, and B. Mohr. The SCALASCA performance toolset architecture. In *International Workshop on Scalable Tools for High-End Computing (STHEC)*, Kos, Greece, June 2008.

[11] G. Griem, L. Oliker, J. Shalf, and K. Yelick. Identifying performance bottlenecks on modern microarchitectures using an adaptable probe. In *Proc. 3rd International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-PDS)*, Santa Fe, New Mexico, Apr. 26-30, 2004.

[12] R. Hedges, B. Loewe, T. McLarty, and C. Morrone. Parallel File System Testing for the Lunatic Fringe: the care and feeding of restless I/O Power Users. In *IEEE NASA Goddard Conference on Mass Storage Systems and Technologies (MSST)*, 2005.

[13] I/O Tips. http://www.nccs.gov/computing-resources/jaguar/debugging-optimization/io-tips/.

[14] The ASCI I/O stress benchmark. http://sourceforge.net/projects/ior-sio/.

[15] MADbench2: A Scientific-Application Derived I/O Benchmark. http://outreach.scidac.gov/projects/madbench/.

[16] M. Mueller and et al. Developing scalable applications with vampir. http://www.vi-hps.org/datapool/page/18/mueller.pdf.

[17] H. Shan, K. Anypas, and J. Shalf. Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark. In *Proc. SC2008: High performance computing, networking, and storage conference*, Austin, TX, Nov 15-21, 2008, to appear.

[18] S. Shende, A. Malony, and D. Cronk. Observing Parallel Phase and I/O Performance Using TAU. In *DoD HPCMP Users Group Conference, 2008*, 14-17 July 2008.

[19] D. Skinner. Integrated Performance Monitoring: A portable profiling infrastructure for parallel applications. In *Proc. ISC2005: International Supercomputing Conference*, volume to appear, Heidelberg, Germany, 2005.

[20] E. Smirni, R. Aydt, A. Chien, and D. Reed. I/O Requirements of Scientific Applications: An Evolutionary View. In H. Jin, T. Cortes, and R. Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, pages 576–594. IEEE Press, Piscataway, New Jersey, 2002.

[21] N. J. Wright, W. Pfeiffer, and A. Snavely. Characterizing parallel scaling of scientific applications using ipm. In *The 10th LCI International Conference on High-Performance Clustered Computing*, March 10-12, 2009.