Optimization Techniques at the I/O Forwarding Layer

Kazuki Ohta,* Dries Kimpe,† Jason Cope,† Kamil Iskra,† Robert Ross,† Yutaka Ishikawa‡
*Preferred Infrastructure, Inc.,
Tokyo, Japan
Email: kzk@preferred.jp
†Argonne National Laboratory, Argonne,
Illinois 60439, USA
Email: {dkimpe, copej, iskra, rross}@mcs.anl.gov
‡Graduate School of Information Science and Technology, The University of Tokyo,
Tokyo, Japan
Email: {kzk@il.is.s, ishikawa@is.s}.u-tokyo.ac.jp

***Abstract*—I/O is the critical bottleneck for data-intensive scientific applications on HPC systems and leadership-class machines. Applications running on these systems may encounter bottlenecks because the I/O systems cannot handle the overwhelming intensity and volume of I/O requests. Applications and systems use I/O forwarding to aggregate and delegate I/O requests to storage systems. In this paper, we present two optimization techniques at the I/O forwarding layer to further reduce I/O bottlenecks on leadership-class computing systems. The first optimization pipelines data transfers so that I/O requests overlap at the network and file system layer. The second optimization merges I/O requests and schedules I/O request delegation to the back-end parallel file systems. We implemented these optimizations in the I/O Forwarding Scalability Layer and them on the T2K Open Supercomputer at the University of Tokyo and the Surveyor Blue Gene/P system at the Argonne Leadership Computing Facility. On both systems, the optimizations improved application I/O throughput, but highlighted additional areas of I/O contention at the I/O forwarding layer that we plan to address.**

***Index Terms*—I/O forwarding; Parallel file systems; Leadership-class machines**

I. Introduction

Current petascale and leadership-class machines, such as the IBM Blue Gene/P and Cray XT systems, consist of several hundreds of thousands of processing elements [1]. Future exascale systems are expected to contain millions of processing elements [2]. While the computational power of supercomputers increases, the I/O system for these machines is often less powerful. Data access rates to hard disks are not improving as quickly as multicore processor computation rates, and the increasing number of processing elements in the systems can generate an overwhelming volume of I/O requests.

As applications continue scaling, additional capabilities are required to support the increased number of I/O requests generated. For example, a potential I/O bottleneck can occur for scientific applications that alternate between computation phases and I/O phases [3] on systems with large imbalances in computation and I/O capabilities. In the I/O phase, applications often write *snapshot* and *checkpoint* data regularly. If a large number of application processes simultaneously enter the I/O phase, the I/O subsystem must handle each application request quickly to ensure high performance. Increases in I/O request volume require additional processing overhead for the I/O system and can decrease the performance of applications and overload the system.

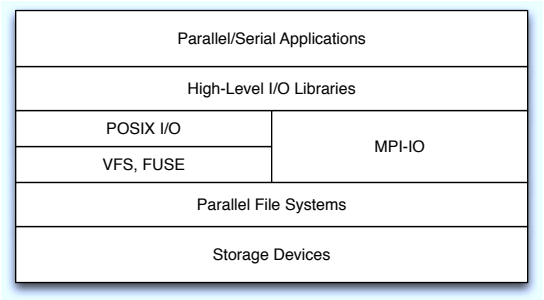

Fig. 1: Current Parallel I/O Software Stack

Current leadership-class systems provide a multilayer software environment to support I/O from scientific applications. Figure 1 shows a multilayered environment that includes high-level scientific I/O libraries [4], [5], parallel I/O using MPI-IO [6], [7], POSIX I/O parallel file systems [8]–[10], and storage devices. While this environment can accelerate parallel application I/O for smaller-scale systems, it does not adequately support the demands of emerging and existing leadership-class systems with large numbers of processing elements.

The goal of I/O forwarding is to eliminate this bottleneck by reducing the number of I/O requests issued to a storage system. Figure 2 illustrates where the I/O forwarding layer exists in the I/O system software stack for HPC environments. With the I/O forwarding layer, all I/O requests are forwarded to dedicated processing elements, known as I/O nodes. When an I/O node receives I/O requests, it redirects them to the back-end parallel file systems. This strategy reduces the number of clients accessing the file systems and can potentially reduce the file system traffic by aggregating and reordering I/O requests.

Several systems, including the IBM Blue Gene platforms, use I/O forwarding to reduce this bottleneck, but they have not addressed additional opportunities to improve I/O forwarding performance. Moreover, these existing I/O forwarding systems are tightly coupled to vendor hardware or software stacks and are not portable to other HPC systems.

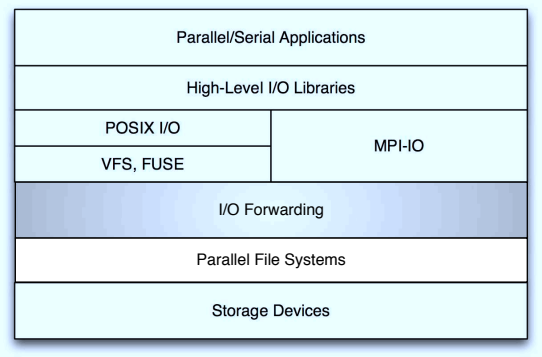

Fig. 2: Parallel I/O Software Stack with I/O Forwarding

The I/O Forwarding Scalability Layer (IOFSL) [11] is a collaborative project that is developing an open-source, portable, high performance I/O forwarding solution for leadership-class systems. In a previous study [12], we described our IOFSL prototype, presented the IOFSL data access interface, and demonstrated the prototype's capabilities on a small Linux cluster. In this paper, we extend that work and focus on implementing and evaluating optimizations at the I/O forwarding layer. One optimization provides an I/O pipeline capability that overlaps network communication and file system I/O requests. Another optimization provides I/O request aggregation techniques to coalesce several I/O requests into fewer requests when possible. The contributions presented in this paper include:

- Description and evaluation of an I/O pipeline mechanism for the I/O forwarding layer that overlaps file system and communication layer I/O requests.
- Description and evaluation of an I/O aggregation, merging, and scheduling mechanism for I/O forwarding systems that reduces the number of independent, noncontiguous file systems accesses by applications.
- Demonstration of a portable, high-performance I/O forwarding layer on an IBM Blue Gene/P system and a Linux cluster.
- Performance comparisons of IOFSL and existing HPC I/O software stacks, including the I/O forwarding infrastructure available on the IBM Blue Gene/P.

In this paper, we describe our recent research and development work for the IOFSL project and present our evaluation of two performance optimizations for IOFSL. Section II describes the goals, architecture, capabilities, and our implemented optimizations for IOFSL. Section III presents the details of these optimizations for IOFSL. Section IV discusses our evaluation and experimental results. Section V summarizes work related to IOFSL. Section VI presents our conclusions and describes future work.

II. I/O Forwarding Scalability Layer

I/O forwarding is an important piece of the leadership-class system I/O stack. Until recently, no portable, open source I/O forwarding implementation was available. Vendors provided I/O forwarding software integrated with a platform's software stack. For Cray XT systems, Cray provides the Cray Data Virtualization Service for I/O forwarding capabilities. The IBM Blue Gene platforms provide I/O forwarding capabilities through the Control and I/O Daemon (ciod). Recently, the I/O Forwarding Scalability Layer project [11] has provided a portable, open-source, scalable I/O forwarding framework for high-performance computing systems. In the following section, we provide a brief overview of the IOFSL project, a description of the current implementation, and highlight potential optimizations to improve IOFSL performance.

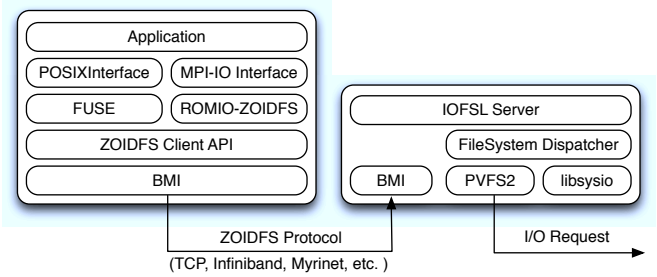

Fig. 3: IOFSL Software Stack

The original IOFSL software was derived from research completed as part of the ZeptoOS I/O Daemon (ZOID) [13] project at Argonne National Laboratory. The ZOID project developed the file system-independent ZOIDFS protocol, function call forwarding infrastructure for the ZeptoOS operating system, and a high-performance transport layer using the Blue Gene/P tree network. IOFSL builds on this work and focuses on developing a portable, high-performance I/O forwarding infrastructure. The IOFSL portability improvements include supporting multiple application I/O interfaces, providing a portable communication layer across multiple network types, integrating with common HPC file systems, and providing portable optimizations to sustain I/O forwarding performance on a variety of HPC systems.

Figure 3 illustrates the current IOFSL architecture. For network portability, IOFSL uses the Buffered Message Interface (BMI) [14]. BMI provides a common interface for data transmission across a variety of networks, including TCP/IP, Myrinet, Infiniband, Cray Portals, and the Blue Gene/P tree network using ZOID. BMI also provides several features that benefit parallel I/O, such as sending noncontiguous buffers in a single network transfer.

Applications communicate I/O requests with IOFSL servers using the ZOIDFS protocol. ZOIDFS provides a file system-independent protocol for forwarding application I/O requests. The protocol is stateless, provides portable and distributable

file handles instead of file descriptors, and provides I/O operations optimized for parallel I/O, such as list I/O read and write operations [15]. XDR is used to encode and decode the ZOIDFS protocol data between the application and the IOFSL server.

IOFSL servers interact with back-end file systems using file system dispatchers that implement the ZOIDFS API. IOFSL issues all file system I/O operations through these dispatchers. The dispatchers use the client interfaces (when available) exposed by some parallel file systems for data access. For example, the PVFS2 IOFSL dispatcher uses the PVFS2 client library to communicate directly with a PVFS2 file system, through the PVFS2 kernel module. Currently, IOFSL provides dispatchers for POSIX compatible file systems, libsysio [16], and PVFS2. We have tested the IOFSL dispatchers on several common HPC file systems, including PVFS2 [8], [17], Lustre [9], and GPFS [10].

IOFSL also provides several application I/O interfaces. Applications using MPICH2 and MPI-IO can leverage the ROMIO ZOIDFS driver. This driver translates MPI-IO requests into the ZOIDFS protocol and can leverage MPI-IO optimizations, such as two-phase collective I/O [6] and data sieving [7]. For applications that use the POSIX I/O, a ZOIDFS FUSE client is available. For systems that use lightweight operating systems or do not support the FUSE client, a libsysio client library with ZOIDFS support is available. To use the libsysio client library, applications compile in the libsysio library with ZOIDFS support. Application POSIX I/O calls are intercepted by the library and translated into equivalent ZOIDFS API invocations. IOFSL support is also available to high-level I/O libraries that can use the IOFSL client interfaces. For example, the ROMIO ZOIDFS driver can be used with the Hierarchical Data Format Version 5 (HDF5) [4] library or the PNetCDF [5] library so that applications using these libraries can communicate with IOFSL.

The IOFSL software stack can be improved in several ways. In this paper, we implement and evaluate the following request aggregation and data transfer optimizations for IOFSL. The IOFSL server uses a thread-based task model to service client I/O requests. As the IOFSL server receives client requests, the requests are converted into tasks. The IOFSL server manages task execution over the lifetime of the client request. The initial IOFSL implementation treated these tasks as independent requests and made no attempt to aggregate request activities or optimize access to the back-end file system. We improved this task model by allowing multiple requests to leverage the ZOIDFS list I/O mechanism and merging adjacent or overlapping requests to reduce the number of file system accesses.

Another improvement focus is on data transfers. The original IOFSL implementation transmits large data chunks between clients and servers. This behavior forces the IOFSL servers and clients to wait for all pending data transmissions before processing any of the transmitted data. We can improve the data transmission between clients and servers by breaking large data segments into smaller data segments and beginning processing the smaller data segments as soon as they are received. This behavior would allow the IOFSL server to overlap I/O operations and reduce blocking I/O requests.

III. Optimizations at the I/O Forwarding Layer

We implemented two optimizations in IOFSL to improve application I/O performance. The first optimization is a pipeline transfer mechanism. The second optimization provides I/O request scheduling for IOFSL. Both optimizations are portable and have several parameters to adjust IOFSL runtime performance. The following subsections describe these optimizations in detail.

A. Pipeline Transfers

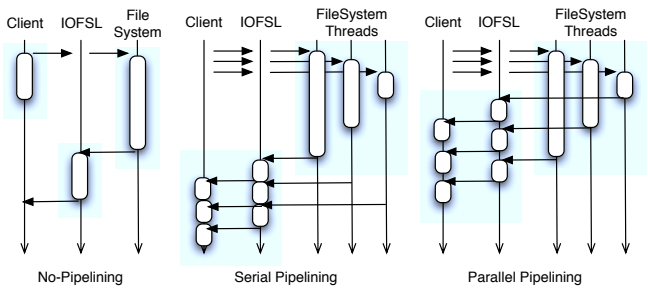

Fig. 4: Pipeline Data Transfer Methods

The goal for the pipeline transfer optimization is to improve data transfer performance between the application and the I/O forwarding servers. Pipeline transfers overlap application I/O requests and file system I/O requests. Overlapping provides two benefits. The first benefit is that the I/O forwarding software can simultaneously post and process multiple data transfers. The second benefit is that data transfers exceeding the amount of memory available to the I/O forwarding software are split into smaller, more manageable chunks of data. This behavior enables processing large data transfers without modification of application or file system I/O behavior.

Multiple methods exist for providing pipeline data transfers. Figure 4 illustrates these methods and describes the behavior of non-pipeline transfers, serial pipeline transfers, and parallel pipeline transfers. The non-pipeline transfer method processes the requested data buffers as contiguous chunks of data. This method does not attempt to split large data requests into smaller chunks. Using the non-pipeline method requires that software components contain the entire data segment before it can begin to process the data segment.

The serial pipeline method allows software components to overlap send and receive operations, but it forces in-order acknowledgments of data operations. By overlapping the I/O operations, the I/O forwarding software can issue multiple asynchronous I/O operations in parallel and monitor the operations for completion. The I/O forwarding layer can use the idle time generated by the nonblocking I/O for additional data processing tasks. The in-order processing ensures that applications or file systems receive data in order, but it can underutilize resources while waiting for the next in-order data

segment to be processed. When the next in-order data segment is not available, the executing I/O operation blocks or stalls the entire processing pipeline until it is available.

The parallel pipeline mode relaxes the in-order acknowledgment constraint of the serial pipeline mode. This approach allows the I/O forwarding server to process and issue I/O requests out of order, thereby preventing active I/O requests from stalling pending I/O requests within the same pipeline while processing resources are available. Although this approach can achieve higher I/O request processing throughput, it requires additional complexity in the I/O forwarding layer to monitor requests and correctly distribute data.

Several parameters can be configured for IOFSL pipelines. One parameter is the size of the pipeline buffer. This parameter is implemented for both the client and the server, so that both software components agree on the maximum pipeline buffer size to transfer and how to split large data streams into smaller chunks. Large pipeline buffers increase network throughput, but the upper limit of the pipeline size is restricted by the BMI transport layer limitations. Another parameter is the size of the IOFSL server memory pool. This parameter limits the number of simultaneous pipeline transfer buffers that can be posted by the IOFSL server. Configuring a large memory pool with many pipeline buffers allows clients to post more simultaneous data transfers, but it may allow IOFSL to consume too much memory on the I/O forwarding node. Overallocating memory may decrease IOFSL performance through excessive paging or may crash I/O forwarding nodes if sufficient memory is not available.

B. Request Scheduler

The goal of request scheduler optimization is to improve I/O accesses between the I/O forwarding server and the global file systems. In the original IOFSL software design, I/O requests are issued as they arrive. This situation can result in many noncontiguous access to the global file system and can reduce application I/O performance. The IOFSL request scheduler aggregates and reorders I/O requests to reduce the number of noncontiguous file system accesses.

Adding the request scheduler to the I/O forwarding layer provides two benefits. The first benefit is that the I/O forwarding scheduler can exploit the global view of parallel applications to sort and merge I/O requests more effectively than an OS kernel currently can. For example, the Linux OS kernel has similar request scheduling capabilities for local file accesses, but it cannot efficiently sort and merge I/O requests because HPC file system drivers often bypass the kernel. The second benefit is that scheduling and merging the requests at the I/O forwarding server requires less overhead than at the parallel file system. The I/O forwarding layer reduces the number of clients that access the file system because the I/O forwarding servers delegate I/O requests for applications. If the I/O forwarding server analyzes all client requests that it will delegate and merges the requests when possible, the server can reduce the total number of I/O requests issued to the file system. Without the request aggregation at the I/O forwarding layer, I/O requests would fill the file system request queues, and there would be no opportunity to merge requests.

As the IOFSL server receives the I/O requests, they are queued in the request scheduler. The request scheduler attempts to merge multiple requests into a single I/O request. After the scheduling interval has elapsed, the scheduler issues the available I/O requests. Since the request scheduler is independent of the file system implementation and decoupled from the transport layer, the request scheduler is portable to systems that support IOFSL.

Arguably, the additional processing from the request scheduler can potentially consume significant computational resources and diminish I/O performance under certain application I/O loads. For example, changing the request scheduling algorithm can affect I/O request throughput and latency depending on how the scheduling algorithm prioritizes requests. The choice of request merging algorithms and data structures can impact the time to issue the merged I/O requests. As the number of requests increases, the cost of request-merging algorithms may become prohibitively expensive. In the following subsections, we describe the algorithms and data structures developed to support the request scheduler and how they affect the request scheduling within the I/O forwarding layer.

1) FIFO and Handle-Based Round-Robin Scheduling: We implemented two request schedulers in IOFSL. The first scheduler uses a first in, first out (FIFO) scheduler that issues I/O requests as it receives them. This approach does not attempt to merge or aggregate requests. The second request scheduler approach uses the handle-based round-robin (HBRR) scheduling algorithm [18], [19]. This scheduler optimizes file system access by merging I/O requests for each file handle using a round-robin scheduling strategy.

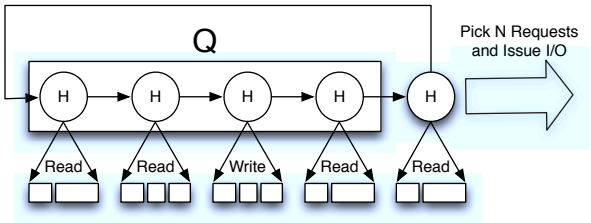

Fig. 5: Handle-Based Round Robin Scheduler

Figure 5 illustrates how the HBRR scheduler functions. The request scheduler has a global queue (called Q) and manages the I/O request for each file handle. When new I/O requests arrive at the scheduler, the scheduler first checks whether the corresponding file handle is in the request queue, Q. If the file handle exists in Q, the scheduler updates the file handle with the new I/O request. If the file handle does not exist in Q, the scheduler adds a new file handle to Q with the new I/O request. Separate entries in the queue are created for read and write operations on the same file handle.

When issuing I/O requests, the scheduler removes requests from the file handles stored in the scheduler request queue, Q, in a round-robin fashion. The number of I/O requests removed

from Q in each round is a configuration parameter referred to as the quantum. At the start of the round, the scheduler pops a number of I/O requests from the first file handle. It does not remove more requests than specified by the quantum. If the quantum was exhausted but has pending I/O requests, the current handle is moved to the back of Q. If all I/O requests are removed from the file handle, the scheduler removes I/O requests from the next handle in Q. This process is repeated until the quantum is exhausted or no requests are left in Q.

The dequeued requests are issued to the parallel file system using the list I/O ZOIDFS interface of the IOFSL file system dispatcher layer. This approach allows IOFSL to issue multiple requests from different clients with one file system call as long as the requests belong to a single file handle read or write operation. This optimization reduces the number of file system operations IOFSL issues to the file system.

2) Brute Force and Interval Tree Request Merging: As the request scheduler assigns requests to file handles in the scheduler queue, the scheduler can optimize I/O accesses by merging noncontiguous requests. Since the requests are separated by file handle and I/O operation type, the scheduler can use the I/O request offset and size data to determine how new I/O requests relate to existing I/O requests and whether the requests can be combined. For example, the scheduler can examine the current I/O requests assigned to a file handle. If a new request is adjacent to or overlaps an existing request, the scheduler can combine the requests to reduce the number of accesses to the file system.

We developed two approaches to merge pending I/O requests at the I/O forwarding layer. The first approach stores the pending requests for a file handle operation in a vector. It attempts to combine requests by comparing the new request to the existing requests for overlapping or adjacent regions. We refer to this approach as the brute force request merging strategy within IOFSL because of the exhaustive search performed for the new request against all pending requests. While this approach involves a simple implementation, the cost of merging many requests becomes prohibitively expensive because of the comparison strategy.

The second request merging approach we developed for IOFSL is the interval tree request merger. This approach uses an interval tree based on an augmented red-black tree [20] for merging adjacent or overlapping requests. The red-black tree provides a balanced tree with O(log n) cost insert, search, and remove operations. The interval tree finds the location of file data based on the I/O request offset and size data at the algorithmic cost of the red-black tree operations. This provides a more efficient merging method for large numbers of I/O requests, but it is a more complex strategy and may be unnecessary for smaller numbers of I/O requests. In this paper, we use the interval tree request merger in our evaluations.

IV. Performance Evaluation

In this section, we present our evaluation of the IOFSL optimizations. We evaluated these optimizations on a Linux cluster and an IBM Blue Gene/P platform. For each platform, we describe the system configuration and the experimental setup and provide an analysis of our observed results.

A. Evaluation on T2K Cluster

TABLE I: Specification of the T2K (University of Tokyo) Cluster

Node Spec	
CPU	AMD Barcelona, 2.3 GHz, 4 Cores
# Sockets	4
Memory	32GB
Local Disk	SATA (Read: 49.52 MB/sec, Write/39.76 MB/sec)
Interconnects	Myrinet 10 Gbps * 2
Ethernet	Intel E1000 (1 Gbps) * 2
OS	RHELS 5.1 (Kernel 2.6.18-53)
glibc	version 2.5
File System	EXT3

IOFSL was evaluated on 32 nodes of the University of Tokyo's T2K research cluster [21]. All nodes in the T2K cluster are connected by a single Myrinet-10G switch. The specification for individual cluster nodes is shown in Table I. The read and write bandwidth of the local disks in the cluster nodes was measured by using Bonnie++ [22].

MPICH2 [23], version 1.1.1p1, was the default MPI used for the T2K cluster experiments. The Myrinet-10G network was used for the parallel benchmark experiments using the MPICH2 ch3:nemesis:mx device. MPICH2 was configured to use ROMIO for MPI-IO support. The IOFSL driver and PVFS2 drivers were enabled within ROMIO.

PVFS2 version 2.8.1, with additional patches for the BMI-MX driver, was used for this evaluation. PVFS2 was configured to use the "directio" TroveMethod. This configuration bypasses the OS buffer cache and our measurements show the actual disk performance. We configured PVFS2 to use the Myrinet-10G network through BMI-MX transport driver. This configuration used four nodes as PVFS2 metadata nodes and data nodes. For these experiments, we set the PVFS2 stripe size to 256 KB.

We deployed IOFSL on one of the nodes in the PVFS2 cluster with an 8 MB pipeline size. IOFSL used the PVFS2 dispatcher for issuing I/O requests to the PVFS2 file system. The PVFS2 dispatcher allows direct access to the PVFS2 file system instead of accessing the file system through the PVFS2 kernel module. This eliminates the extra memory copies through the I/O path. We used the remaining 28 nodes for running applications.

1) IOR Benchmark: We measured the performance characteristics of IOFSL for various message sizes using the IOR benchmark. In this experiment, 128 processes concurrently write and read data into the same file. The ROMIO-PVFS2 driver and the ROMIO-IOFSL driver were used for applications to communicate with the IOFSL server. IOFSL dispatched the requests to the backend PVFS2 file system. The FIFO and HBRR request schedulers were also used in this evaluations.

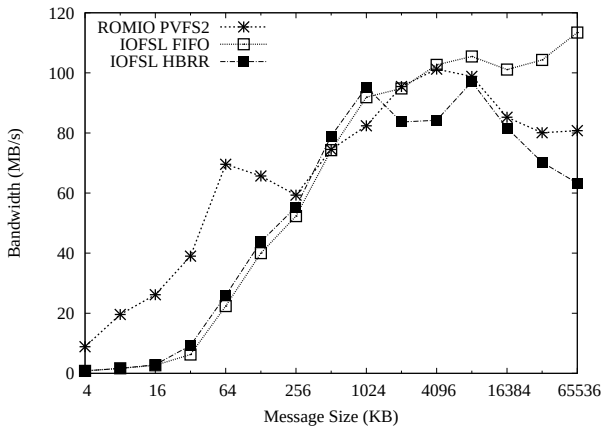

Fig. 6: IOR Benchmark on T2K Tokyo, 128 Processes (Write)

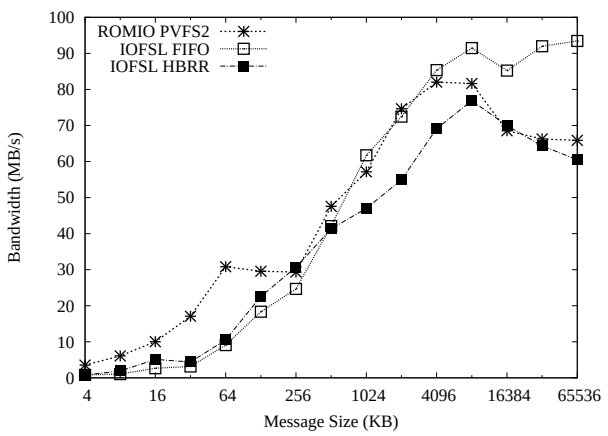

Fig. 7: IOR Benchmark on T2K Tokyo, 128 Processes (Read)

Figures 6 and 7 show the results for this experiment. For large messages (over 8 MB), FIFO performs the best. The FIFO scheduler had a 28.8% write improvement and 29.5% read improvement when compared to PVFS2 with a 64 MB block size.

For small messages, HBRR performs better than the FIFO scheduler. Since HBRR issues list I/O requests, it is able to improve the performance for small messages. For 32KB message sizes, the improvement is 48.6% for writes and 40.0% for reads. For large messages, FIFO outperforms HBRR. HBRR performance is also worse than PVFS2 for some cases. Thus, processing many I/O requests within the IOFSL request scheduler becomes a bottleneck.

While we confirmed that list I/O was effective in IOFSL, request merging did not have a significant impact. We believe the reason is that merging of contiguous requests rarely happens in the IOR workload. In this workload, each process issues contiguous requests, but I/O calls are synchronous. The next request is issued after the previous request is done. When the second request arrives, IOFSL has already issued the previous request and it is no longer in the scheduler's queue. For I/O workloads that share the same I/O pattern evaluated in this experiment, IOFSL request merging will not be able to combine the subsequent I/O requests from the same process.

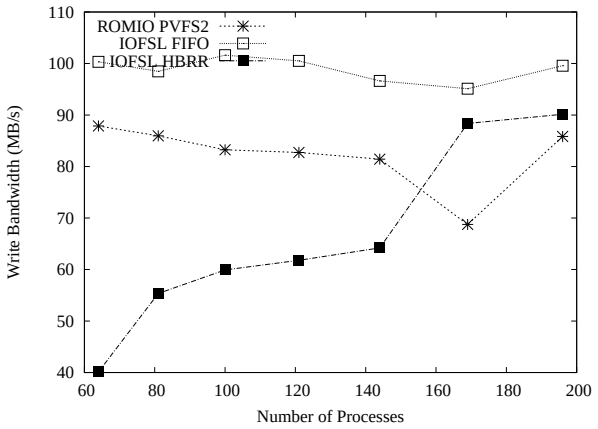

Fig. 8: BTIO Benchmark on T2K Tokyo

2) NAS BTIO: In the next experiment, we measured the scalability of IOFSL for various application processes to IOFSL server ratios using the block-tridiagonal benchmark (BTIO) [24]. BTIO is included in the NAS Parallel Benchmark suite, version 3.3. This benchmark evaluates the write throughput of parallel I/O for various process counts. Each BTIO process is responsible for multiple Cartesian subsets of the entire data set, whose number increases as the square root of the number of processors participating in the computation. BTIO provides three options for how to issue the requests: MPI-IO with collective I/O, noncollective MPI-IO, or Fortran POSIX I/O. Collective I/O was used for this experiment.

BTIO provides different input problem sizes (A-D). We used class C, an aggregate 6.8 GB write. The amount of data written by each process decreases as the number of processes increases because the aggregate write amount is fixed. The BTIO workload uses a common scientific application I/O pattern where the compute phase and the I/O phase alternate.

We measured the I/O bandwidth for BTIO using IOFSL and adjusted the number of processes. Figure 8 presents our results for this experiment. In these tests, IOFSL using the FIFO scheduler performs better than direct PVFS2 access (14.2% to 38.3% improvements). Since this test used collective I/O, IOFSL request merging did not improve performance. This result is similar to the results with large-message cases in the IOR benchmark.

B. Evaluation on Blue Gene/P

The optimizations presented in this paper were also evaluated on the Surveyor IBM Blue Gene/P system at the Argonne Leadership Computing Facility (ALCF). Surveyor is a 4,096-core research and development Blue Gene/P platform. Surveyor's storage system consists of four file servers running PVFS and a DataDirect Networks S2A9550 SAN. Surveyor contains 16 partitions consists of 64 quad-core compute nodes. Each partition has a dedicated quad-core I/O node (ION). Compute nodes forward system calls to the ION over the Blue

Gene/P tree network when using the default IBM software stack. Figure 9 shows the I/O architecture of the ALCF BG/P systems.

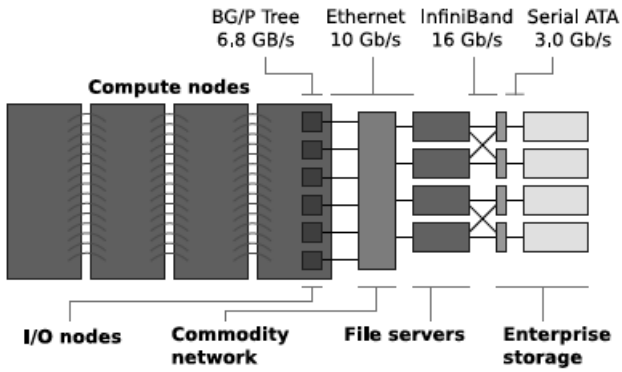

Fig. 9: IBM Blue Gene/P I/O System

When using the default IBM software stack on Surveyor, the compute nodes running the IBM Compute Node Kernel (CNK) forward system calls over the Blue Gene/P tree network to the IBM Control and I/O Daemon (ciod) running on the ION. Since the file I/O system calls are forwarded to the ION, ciod also provides I/O forwarding capabilities for applications running on the Blue Gene/P.

In the ZeptoOS environment it is not possible to use ciod. When we used the ZeptoOS [25] compute node kernel and I/O node kernel, we used the IOFSL software stack to provide I/O forwarding support. When IOFSL is used on Surveyor or other Blue Gene/P systems, a BMI driver for ZOID is available as a high performance alternative to the TCP/IP BMI driver. The ZOID BMI driver implements BMI's device layer API using the Blue Gene/P tree network. Figure 10 illustrates the performance of the BMI drivers and Blue Gene/P network throughput on Surveyor using a BMI ping-pong benchmark.

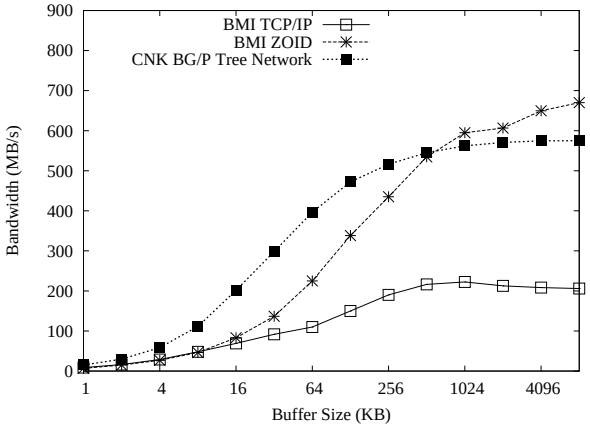

Fig. 10: BMI PingPong Performance on Surveyor

1) Comparison with ciod: We compared IOFSL with ciod using the IOR benchmark. For these experiments, we used 256 compute nodes and 4 I/O nodes. One IOR process was run on each compute node in these experiments. IOFSL and ciod accessed the back-end PVFS2 file servers using the PVFS2 kernel module. Unlike the T2K cluster experiments, IOFSL used the POSIX file system dispatcher. Since the write throughput measurements from this experiment yielded similar results to the read measurements, we focus our evaluation on the IOR read measurements, as shown in Figure 11.

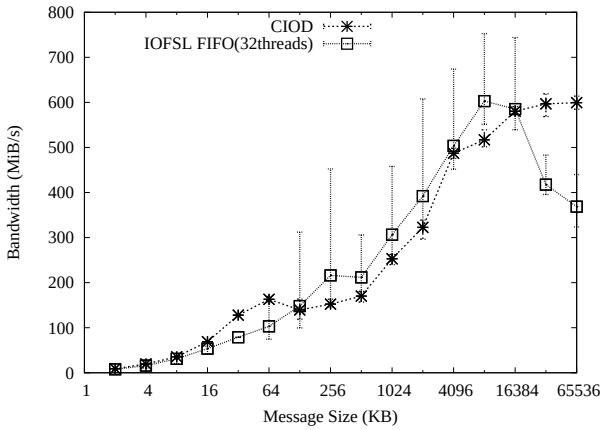

Fig. 11: IOR Benchmark on Surveyor (Read)

For the 128 KB to 16 MB tests, IOFSL performs better than ciod. The performance improvement is as high as 42.0% for the 256 KB test case. For larger test cases (64 MB buffers and larger), we observed a 38.5% degradation in IOFSL performance compared to ciod. During the 64 MB tests, we observed high computation loads for the ION system CPUs. Because of these processing loads, we believe that IOFSL processing on the ION is currently the bottleneck for large buffers. IOFSL also exhibits considerable performance variations, as can be seen in the large error bars plotted in Figure 11 for IOFSL. The throughput for the initial IOR benchmark iteration was much larger than for subsequent iterations.

Currently, the IOFSL server uses a thread-based model for managing I/O requests. The experimental results presented in Figure 11 were conducted with a thread pool consisting of 32 threads. Since the number of threads is larger than the number of cores on the ION (4 cores), we believe thread contention negatively affects IOFSL performance.

Using the same IOR benchmark setup, we evaluated the impact of IOFSL thread pool size on I/O performance. Figure 12 shows the results of this experiment. Considerable performance degradation occurs for the large messages with a thread pool of size 32. The 16-thread-pool case had less performance degradation than the 32-thread-pool case. When comparing the results in Figure 12 to the results in Figure 11, the 16 thread pool case performs as well as ciod for message sizes up to 32 MB. For write operations (Figure 13), the 16 thread pool case shows severe performance degradation in small messages up to 8 MB. For data transfers 8 MB and larger, IOFSL uses pipeline data transfers and this appears to provide a boost for

the 16-thread-pool case.

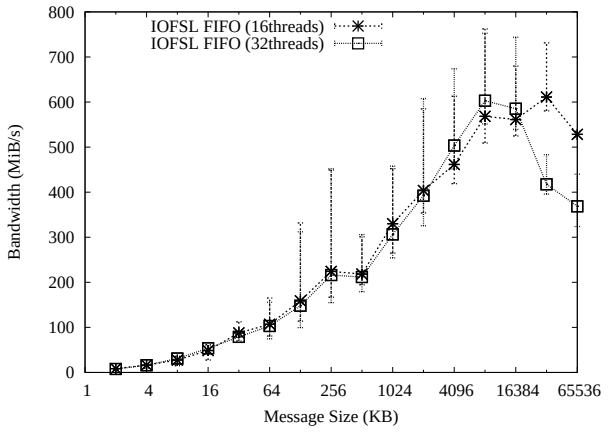

Fig. 12: The Impact of IOFSL Thread Count on Surveyor (Read)

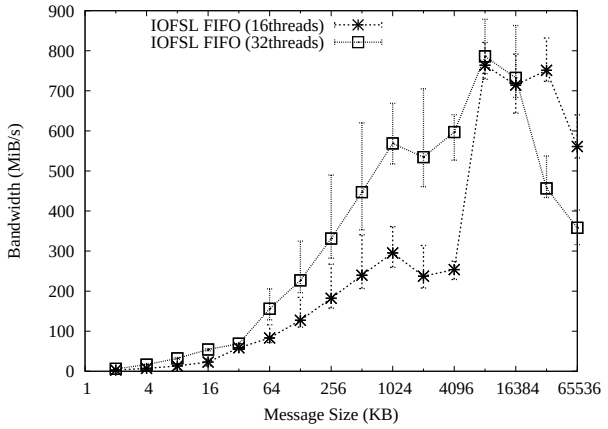

Fig. 13: The Impact of IOFSL Thread Count on Surveyor (Write)

To address this problem, we developed an event-driver IOFSL server. This new server uses a thread pool with a thread count that matches the number of available cores. Instead of treating the I/O requests and IOFSL tasks as single units of work, we developed a series of state machines with many smaller units of work. These smaller units of work execute when computational resources are available and use a callback mechanism to advance the execution of a state machine for the I/O request. We believe this architecture will improve the thread contention issues we observed on the Blue Gene/P I/O nodes. Our current research focuses on evaluating this new server architecture against the thread-model discussed in this paper.

V. Related Work

Several examples exist of I/O forwarding services similar to IOFSL. The Computational Plant (Cplant) [26] at Sandia National Laboratories used an I/O forwarding layer. The Cplant compute nodes forwarded the I/O requests to *yod* servers, which performed the actual I/O on behalf of the compute nodes. The IBM Blue Gene series uses I/O forwarding to ship I/O operations from compute nodes to dedicated I/O nodes [27]. I/O operations from the compute nodes are shipped to a dedicated I/O node over a collective network. The ciod daemon running on the I/O nodes performs I/O to the back-end parallel file systems. Cray has a similar service for the Cray XT series of computers, called the Cray Data Virtualization Service. ZOID [13] is an open-source I/O forwarding framework for IBM Blue Gene/P. ZOID is tightly coupled to the Blue Gene series of supercomputers and is not portable to other HPC systems, such as the Cray XT series or Linux clusters. Condor [28] provides a remote system call interface that allows users to issue remote I/O calls.

Several studies evaluate scheduling algorithms at the parallel file systems layer [29], [30]. Qian et al. proposed a network request scheduler for a large-scale Lustre parallel file system [19]. They describe a quantum-based and object-based round-robin scheduling algorithm to reorder the I/O requests per data object at the server side. Their simulation results show that the scheduling algorithm increases the I/O performance up to 40%.

Gather-arrange-scatter [31] is a node-level request scheduling scheme intended to improve parallel write performance. Using this method, applications issue I/O requests asynchronously, and an intermediate server buffers them. If the length of the buffer exceeds a specified limit, then the buffered requests are reordered, merged, and scattered to the back-end parallel file systems. This scheme enables the intermediate servers to aggregate the requests from the same application process. The method improves the write performance, but it is not portable to applications that do not support asynchronous I/O.

VI. Conclusion

I/O is the critical bottleneck for data-intensive scientific applications in HPC systems. I/O forwarding attempts to bridge the gap between computation and I/O by regulating the file system traffic. Performance optimizations are necessary at the I/O forwarding layer in order to sustain high-performance I/O. In this paper, we propose several optimization techniques for the I/O forwarding layer. The first is a pipeline transfer, which enables the server to overlap the receiving and sending of the data buffers. The second is a request scheduler, which aggregates and reorders the I/O requests for optimized I/O to the back-end parallel file system.

IOFSL was evaluated on two systems. We first evaluated on 32 nodes of the T2K Tokyo Research Cluster. For large requests using the IOR benchmark, IOFSL improves the performance up to 28.8% for writes and up to 29.5% for reads. For small requests, the scheduler issues list I/O requests from individual requests and improves performance 28.8% for writes and 29.5% for reads for 32 KB messages. With the BTIO benchmark, IOFSL performs better than direct access to PVFS2: we observed improvements of 14.2% to 38.3%.

These results show that I/O forwarding is effective in cluster environments.

IOFSL was also evaluated on the ALCF Blue Gene/P environment. IOFSL shows considerable performance improvements over IBM's default I/O forwarding implementation. For 256 KB reads, we observed a performance improvement of 42.0%. However, we observed performance degradation in IOFSL for especially large messages, and it was revealed that thread contention within the IOFSL server affects performance.

We have identified several areas of future work. One area we are evaluating is to an event-driven server architecture. Currently, IOFSL I/O requests are represented as individual threads. We believe this model is the cause of thread contention in the IOFSL server. The new server architecture uses a thread pool, smaller units of work, and a callback mechanism to limit the number of threads executing simultaneously and reduce competition for resources. We are also developing a collaborative caching layer for IOFSL. This layer provides a temporal cache for accessing data shared across the application or temporarily storing pending data requests. Furthermore, we are evaluating IOFSL on additional systems including the Jaguar Cray XT4 and XT5 systems at the Oak Ridge Leadership Computing Facility.

Acknowledgments

This work was supported by the Office of Advanced Scientific Computer Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357. The IOFSL project is supported by the DOE Office of Science and National Nuclear Security Administration (NNSA). 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.

References

[1] "Top500 list," http://www.top500.org/. [Online]. Available: http://www.top500.org/

[2] V. S. ed., "ExaScale Software Study: Software Challenges in Extreme Scale Systems," September 2009.

[3] E. Smirni and D. A. Reed, "Lessons from Characterizing the Input/Output Behavior of Parallel Scientific Applications," *Perform. Eval.*, vol. 33, no. 1, pp. 27–44, 1998.

[4] "HDF5," http://www.hdfgroup.org/HDF5/.

[5] J. Li, W. keng Liao, A. N. Choudhary, R. B. Ross, R. Thakur, W. Gropp, and R. Latham, "Parallel netCDF: A Scientific High-Performance I/O Interface," *CoRR*, vol. cs.DC/0306048, 2003.

[6] J. M. del Rosario, R. Bordawekar, and A. Choudhary, "Improved Parallel I/O via a Two-Phase Run-time Access Strategy," *SIGARCH Comput. Archit. News*, vol. 21, no. 5, pp. 31–38, 1993.

[7] R. Thakur, W. Gropp, and E. Lusk, "Optimizing noncontiguous accesses in MPI – IO," *Parallel Comput.*, vol. 28, no. 1, pp. 83–105, 2002.

[8] P. H. Carns, W. B. L. III, R. B. Ross, and R. Thakur, "PVFS: A Parallel File System for Linux Clusters," in *ALS'00: Proceedings of the 4th Annual Linux Showcase & Conference, Atlanta*. Berkeley, CA: USENIX Association, 2000, pp. 28–28.

[9] "Lustre file system," http://lustre.org/.

[10] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," in *Proceedings of the First Conference on File and Storage Technologies (FAST)*, Jan. 2002, pp. 231–244. [Online]. Available: citeseer.ist.psu.edu/schmuck02gpfs.html

[11] "IOFSL: I/O Forwarding and Scalability Layer," http://www.iofsl.org/.

[12] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan, "Scalable I/O Forwarding Framework for High-Performance Computing Systems," in *IEEE Int'l Conference on Cluster Computing (Cluster 2009)*, September 2009.

[13] K. Iskra, J. W. Romein, K. Yoshii, and P. Beckman, "ZOID: I/O-Forwarding Infrastructure for Petascale Architectures," in *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: ACM, 2008, pp. 153–162.

[14] P. Carns, R. Ross, W. Ligon, and P. Wyckoff, "BMI: A Network Abstraction Layer for Parallel I/O," in *In Proceedings of IPDPS05, CAC Workshop*, 2005.

[15] R. Thakur, W. Gropp, and E. Lusk, "On implementing MPI-IO portably and with high performance," in *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, 1999, pp. 23–32. [Online]. Available: citeseer.ist.psu.edu/article/thakur99implementing.html

[16] "libsysio," http://sourceforge.net/projects/libsysio.

[17] T. P. Community, "Parallel virtual file system, version 2," http://www.pvfs.org/.

[18] K. Ohta and Y. Ishikawa, "Scalable Parallel I/O Systems by Client-Side Optimizations," Master's thesis, University of Tokyo, March 2010.

[19] Y. Qian, E. Barton, T. Wang, N. Puntambekar, and A. Dilger, "A novel network request scheduler for a large scale storage system," *Computer Science - R&D*, vol. 23, no. 3-4, pp. 143–148, 2009.

[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*. The MIT Press, 2009.

[21] "T2K," http://www.open-supercomputer.org/.

[22] "Bonnie++," http://www.coker.com.au/bonnie++/.

[23] "MPICH2: High-performance and Widely Portable MPI," http://www.mcs.anl.gov/research/projects/mpich2/.

[24] P. Wong and R. F. V. der Wijngaart, "NAS Parallel Benchmark I/O Version 2.4, NAS Technical Report nas-03-002, NASA Ames Research Center, Moffett Field, CA 94035-1000."

[25] "Zeptoos," http://www.mcs.anl.gov/research/projects/zeptoos/.

[26] Sandia National Laboratories, "Computational Plant," http://www.cs.sandia.gov/cplant.

[27] H. Yu, R. K. Sahoo, C. Howson, G. Almasi, J. G. Castanos, M. Gupta, J. E. Moreira, J. J. Parker, T. E. Engelsiepen, R. Ross, R. Thakur, R. Latham, and W. D. Gropp, "High performance file I/O for the Blue Gene/L supercomputer," in *International Symposium on High-Performance Computer Architecture*, Feb. 2006.

[28] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: The Condor experience." *Concurrency - Practice and Experience*, vol. 17, no. 2-4, pp. 323–356, 2005.

[29] R. Ross, "Reactive Scheduling for Parallel I/O Systems," Ph.D. dissertation, Clemson University, 2000.

[30] A. Lebre, G. Huard, Y. Denneulin, and P. Sowa, "I/O Scheduling Service for Multi-Application Clusters," in *Proceedings of IEEE Cluster 2006*, 2006.

[31] K. Ohta, H. Matsuba, and Y. Ishikawa, "Improving parallel write by node-level request scheduling," in *IEEE Int'l Symposium on Cluster Computing and the Grid (CCGrid 2009)*, May 2009.