Scalable I/O Forwarding Framework for High-Performance Computing Systems

NAWAB ALI, PHILIP CARNS, KAMIL ISKRA, DRIES KIMPE, SAMUEL LANG, ROBERT LATHAM, ROBERT ROSS, LEE WARD, AND P. SADAYAPPAN

Technical Report
OSU-CISRC-4/09-TR07

Scalable I/O Forwarding Framework for High-Performance Computing Systems

Nawab Ali,* Philip Carns,† Kamil Iskra,† Dries Kimpe,‡ Samuel Lang,† Robert Latham,† Robert Ross,†
Lee Ward,§ P. Sadayappan*
*The Ohio State University
Email: {alin, saday}@cse.ohio-state.edu
†Argonne National Laboratory
Email: {carns, iskra, slang, robl, rross}@mcs.anl.gov
‡University of Chicago
Email: dkimpe@mcs.anl.gov
§Sandia National Laboratories
Email: lee@sandia.gov

***Abstract*—Current leadership-class machines suffer from a significant imbalance between their computational power and their I/O bandwidth. While Moore's law ensures that the computational power of high-performance computing systems increases with every generation, the same is not true for their I/O subsystems. The scalability challenges faced by existing parallel file systems with respect to the increasing number of clients, coupled with the minimalistic compute node kernels running on these machines, call for a new I/O paradigm to meet the requirements of data-intensive scientific applications. I/O forwarding is a technique that attempts to bridge the increasing performance and scalability gap between the compute and I/O components of leadership-class machines by shipping I/O calls from compute nodes to dedicated I/O nodes. The I/O nodes perform operations on behalf of the compute nodes and can reduce file system traffic by aggregating, rescheduling, and caching I/O system calls. This paper presents an open, scalable I/O forwarding framework for high-performance computing systems. We describe an I/O protocol and API for shipping function calls from compute nodes to I/O nodes, and we present a quantitative analysis of the overhead associated with I/O forwarding.**

I. Introduction

Current leadership-class machines such as the IBM Blue Gene/P supercomputer at the Argonne National Laboratory [1] and the Cray XT system at the Oak Ridge National Laboratory [2] consist of a few hundred thousand processing elements. Future generations of supercomputers will incorporate millions of processing elements. This significant increase in scale is brought about not only by an increase in the number of nodes, but also by new multicore architectures that can accommodate an increasing number of processing cores on a single chip.

While the computational power of supercomputers keeps increasing with every generation, the same is not true for their I/O subsystems. The data access rates of storage devices have not kept pace with the exponential growth in microprocessor performance, meaning that an increasingly large number of I/O devices are needed to provide corresponding I/O rates. This situation has adversely affected the I/O bandwidth-to-flops (floating-point operations per second) ratio of these systems. While the I/O bandwidth of earlier supercomputers was around 1 GBps for every Teraflop [3], [4], the I/O bandwidth-to-flops ratio of current leadership-class machines is around 1 GBps for 10 Teraflops [1]. I/O is a bottleneck for an increasing number of applications, and it has the potential of critically impacting application performance on the next generation of supercomputers.

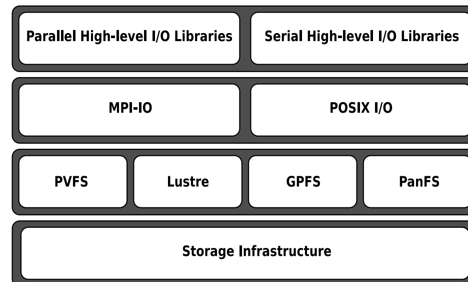


Fig. 1: Typical I/O software stack for HPC systems.

Given the limitations imposed by current storage hardware technology, the main challenge facing I/O researchers is to drive the existing I/O infrastructure at maximum efficiency while simultaneously scaling to a larger number of processing elements. Figure 1 shows the I/O software stack available on a typical high-performance computing (HPC) system. It consists of serial and parallel high-level I/O libraries, MPI-IO and POSIX I/O implementations, parallel file systems, and the storage infrastructure. An important question that needs to be answered is, where in the software stack do we make improvements so as to have the greatest impact on application performance?

Parallel file systems are an obvious potential target for improvements. The file systems available on current leadership-class machines, such as PVFS [5], GPFS [6], Lustre [7], and

This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

PanFS [8] were designed with smaller systems in mind. They face significant challenges scaling to the hundreds of thousands of clients that are available on current high-performance computing systems. Moreover, since not all HPC systems use the same parallel file system, attempting to address this challenge at the file system layer might prove ineffective.

Another option is to make the scalability improvements at the MPI-IO layer. ROMIO [9], from Argonne National Laboratory is the *de facto* standard MPI-IO implementation. Since it is distributed as part of both OpenMPI and MPICH2 libraries, it is available on most HPC systems. However, not all applications use the MPI-IO interface for I/O, so any improvements made at the MPI-IO layer may not be visible to the entire spectrum of scientific applications. Similarly, parallel high-level libraries such as Parallel-NetCDF [10] are not used widely enough to have adequate impact. POSIX implementations and serial high-level libraries are an artifact from an earlier generation and are available on current HPC systems only to support legacy applications.

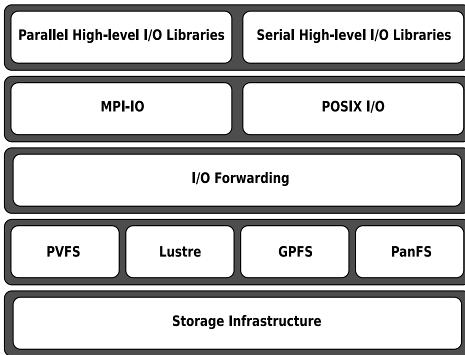


Fig. 2: I/O software stack with I/O forwarding.

Some HPC systems run a minimalistic operating system kernel on the compute nodes to limit the operating system (OS) "noise". The IBM Blue Gene series of supercomputers goes even further, restricting I/O operations from the compute nodes. To enable applications to perform I/O, the compute node kernel ships all I/O operations to a dedicated I/O node, which performs I/O on behalf of the compute nodes. This concept, known as *I/O forwarding*, is explained in detail in Section II. As shown in Figure 2, I/O forwarding introduces a new layer in the I/O software stack. By interposing software above the file system layer but below the rest of the I/O software stack, the I/O forwarding framework provides a compelling point for I/O optimizations. It is transparent to applications and high-level I/O libraries and any optimization performed at the I/O forwarding layer is visible across all parallel file systems.

In view of the importance of I/O forwarding in HPC systems, it is desirable to have a high-quality implementation capable of supporting multiple architectures, file systems, and high-speed interconnects. While a few I/O forwarding solutions are available for the IBM Blue Gene platform, they are tightly coupled to that architecture [11], [4].

In this paper we present a scalable I/O forwarding framework (IOFSL) for high-performance computing systems. We describe a new protocol and API (ZOIDFS) for forwarding I/O function calls from the compute nodes to the I/O node, and we quantify the overheads associated with introducing the I/O forwarding layer in the I/O stack.

The rest of the paper is organized as follows. Section II describes I/O forwarding. Section III presents our scalable I/O forwarding framework. We describe the components of the I/O forwarding software stack in Section IV. Section V discusses the experimental results. We summarize the prior research in I/O forwarding in Section VI and present our conclusions and future work in Section VII.

II. I/O Forwarding

General-purpose operating systems such as Linux are designed for a multiuser and multiprogramming environment. They employ mechanisms such as multitasking, process preemption, and context switching to ensure a low response time for applications. While these mechanisms fulfill the requirements of desktop and server environments, they also introduce significant levels of noise in the system in the form of context switches, cache poisoning, translation lookaside buffer (TLB) misses, and interrupts. Operating system noise can have an adverse impact on the performance of HPC applications, particularly with respect to synchronicity [12], [13].

An important component of a scalable system is system software with low noise. A number of efforts have addressed the issue of system noise by deploying lightweight kernels [14], [15] and through tuning of the Linux kernel [16]. File system software can be a significant contributor to system noise. File systems that cache locks or dirty data can initiate asynchronous communication that can significantly skew application synchronicity. Removing these file system components from the compute nodes eliminates one source of noise.

To mitigate the levels of noise in OS kernels, massively parallel machines such as the IBM Blue Gene/P run customized, stripped-down versions of the OS kernels on the compute nodes. The Blue Gene/P compute node kernel (CNK) is a lightweight kernel that minimizes OS interference by disabling support for multiprocessing and POSIX I/O system calls [17].

There are several ways to provide I/O support on the Blue Gene/P. Applications can use a user-level file system based on FUSE [18] or the SYSIO library [19] to perform file I/O. The SYSIO library provides POSIX-like file I/O support for remote file systems in userspace. Another approach, which is used by the Blue Gene architecture, is to forward all I/O requests from the compute nodes to dedicated I/O nodes. The I/O nodes run a fully functional OS kernel and perform I/O on behalf of the compute nodes. This technique, known as I/O forwarding, enables applications running on the compute nodes to perform I/O without introducing I/O-specific jitter in the CNK.

Figure 3 shows the I/O forwarding infrastructure on the IBM Blue Gene/P. Compute nodes are partitioned into subsets that map to an I/O node. The ratio of I/O nodes to compute nodes

varies from 1:8 to 1:64. The compute nodes are connected to the I/O nodes via a collective tree network. The I/O nodes are connected to the file system via a 10 GigE network. The CNK forwards all I/O and socket requests to the I/O node. A dedicated control and I/O daemon (CIOD) running on the I/O node performs I/O on behalf of the compute nodes by invoking the corresponding file system calls.

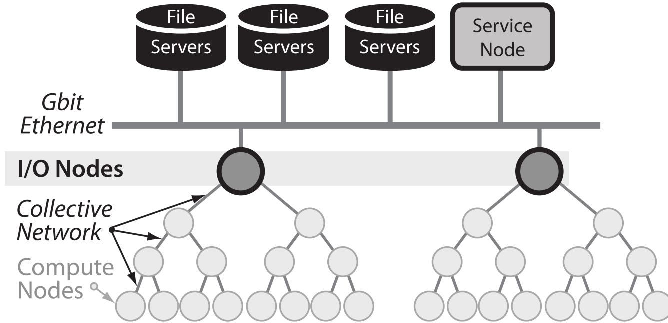


Fig. 3: I/O forwarding architecture for IBM BG/P.

Most current file systems track all active clients. While this was not a significant issue with hundreds or even thousands of clients, today's largest systems consist of hundreds of thousands of processing elements, and the overhead of this tracking places enormous burdens on file systems. However, by partitioning the compute nodes into *M* subsets, each containing *N* compute nodes, and by forwarding the I/O requests from each subset to a dedicated I/O node, we can reduce by a factor of *N* the apparent number of clients accessing the file system. I/O forwarding can potentially reduce the file system traffic by aggregating, rescheduling, and caching the I/O requests at the I/O nodes. These optimizations are relevant even for architectures that provide connectivity between compute nodes and file system servers, such as the Cray XT and Linux clusters.

I/O forwarding also allows the compute nodes to bridge the multiple physical networks available on the Blue Gene/P. Since the compute nodes are not directly connected to the file servers, they ship their I/O requests over the collective tree network to the I/O nodes, thus circumventing the lack of direct connectivity to the file system.

There are some significant drawbacks associated with the Blue Gene/P I/O forwarding implementation. The CNK supports only a subset of the POSIX I/O and BSD socket API. Applications using MPI-IO need to translate the MPI-IO calls to POSIX I/O, which eliminates the possible use of file system specific optimizations performed at the MPI-IO layer. Also, the BG/P I/O forwarding infrastructure is tightly coupled to IBM technologies.

III. Scalable I/O Forwarding Framework

In view of the importance of I/O forwarding in the I/O stack of leadership-class machines and the lack of a portable, open-source implementation, we propose a scalable, unified I/O forwarding framework for high-performance computing systems. In particular, this layer will perform the following functions:

- Provide function shipping at the file system interface level that enables asynchronous coalescing and I/O without jeopardizing determinism for computation
- Offload file system functions from simple or full OS client processes to multiple targets, including another core or hardware on the same system, an I/O node on a conventional cluster, or a service node on a leadership-class system
- Reduce the number of file system operations or clients that are visible to the file system
- Support multiple parallel file systems
- Support multiple high-speed interconnects and networking solutions
- Integrate with MPI-IO and any hardware features designed to support efficient parallel I/O

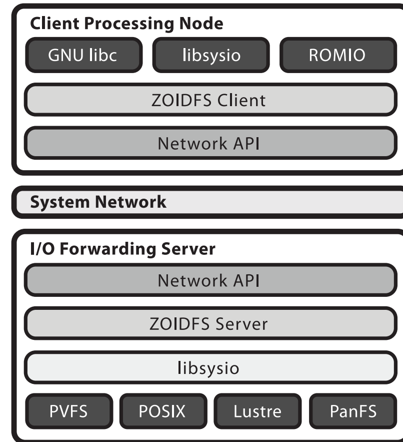


Fig. 4: I/O forwarding software stack.

Our I/O forwarding framework leverages previous work from the ZOID project at Argonne National Laboratory [4]. In particular, we use the ZOIDFS I/O protocol and API as a starting point for our research. Figure 4 shows the I/O forwarding software stack. It consists of two main components: a ZOIDFS client library running on the compute nodes and an I/O forwarding daemon (IOD) running on I/O nodes. The ZOIDFS client library forwards I/O requests from the compute node kernel to the IOD. The IOD performs file I/O on behalf of the compute nodes by executing the corresponding file system calls.

One of the design requirements of the I/O forwarding framework was portability. We did not want to make any assumptions about operating system kernels, high-speed interconnects, file systems, or machine architectures that the framework would operate on. In view of the above design requirement, we have introduced abstractions at the client, network, and file system layers.

We encode the function parameters using XDR [20] to account for possible heterogeneity between the compute node and I/O node architectures. Similarly, through the abstraction

provided by the SYSIO library, we can potentially support multiple file systems on the I/O nodes, including PVFS, Lustre, UFS, and PanFS. We use BMI [21] to provide our network abstraction, allowing us to operate over several high-speed interconnects such as InfiniBand [22], Myrinet, and Gigabit Ethernet. Section IV discusses these components in greater detail.

IV. I/O Forwarding Software Stack

This section describes the individual components of the I/O forwarding software stack.

A. ZOIDFS I/O Protocol

The POSIX file I/O protocol inhibits the performance of file systems in the HPC domain [23], [24]. To avoid the associated performance overhead, stateless file systems such as PVFS relax certain POSIX consistency semantics [25]. Also, the POSIX API is not expressive enough to concisely describe I/O patterns such as noncontiguous file access, that are often present in high-performance computing applications. POSIX requires extensions to enable efficient noncontiguous I/O.

To overcome the limitations of POSIX file I/O, we have defined a new I/O protocol, called ZOIDFS, that is suitable for the I/O forwarding framework. ZOIDFS is a stateless protocol. Instead of file descriptors, it uses opaque file handles to describe the objects on which I/O operations will be performed. Since the protocol does not maintain any state at the client or the server end, these handles can be freely exchanged among the compute nodes.

The ZOIDFS API is more expressive than POSIX and requires fewer I/O calls for file operations. For instance, ZOIDFS has no file open or close calls. Instead, applications perform a file lookup to obtain the file handle, and all subsequent operations use the file handle to perform I/O. The ZOIDFS I/O calls can operate on multiple memory buffers and regions of the file using a single call. This approach allows applications with noncontiguous I/O patterns to effectively perform I/O without the overhead of invoking multiple read and write calls.

```
int zoidfs_lookup(const zoidfs_handle_t *parent_handle,
                  const char *component_name,
                  const char *full_path,
                  zoidfs_handle_t *handle);

int zoidfs_write(const zoidfs_handle_t *handle,
                 size_t mem_count,
                 const void *mem_starts[],
                 const size_t mem_sizes[],
                 size_t file_count,
                 const uint64_t file_starts[],
                 uint64_t file_sizes[]);
```

While the ZOIDFS API is feature complete and stable, we have identified the need to pass *hints* along with the API function calls, to provide contextual information helpful for optimizations or debugging. Potential parameters that can be passed as hints include *node id*, *process id*, *operation id*, and *user credentials*. For instance, the *operation id* can identify individual suboperations coming from multiple compute processes that form a larger, application-wide collective operation. This information can be helpful to a separate caching layer running on the I/O forwarding nodes. We are currently exploring extensions to the ZOIDFS API to include the *hints* parameter.

The stateless nature of the ZOIDFS protocol introduces important security challenges. The typical approach to security in a system like this would be to track clients from open to close and associate credentials with the tracked connection. Tracking potentially hundreds of thousands of clients is exactly the sort of problem we are hoping to avoid with our solution. Instead of tracking individual client processes, we note that processes are merely part of a larger application executing as some user. We plan to pass credentials with client requests and to incorporate capabilities into file handles in a way that preserves well-understood security models but allows for significant reduction in operations and security-related traffic.

B. ZOIDFS Client Interface

The ZOIDFS API is intended to be used within ROMIO [9] for MPI-IO [26] and under FUSE [18], SYSIO [19], or glibc wrappers for POSIX. The ROMIO driver for ZOIDFS enables parallel applications to perform I/O call forwarding via MPI-IO. It converts MPI file views and datatypes into offset-list pairs that can be serviced with a single, noncontiguous ZOIDFS operation. We can utilize other ROMIO optimizations as well, such as two-phase collective I/O and data sieving (for reads only, as writes would require locking).

The ROMIO support on the Blue Gene/P system further illustrates the limitations of POSIX in high-performance computing environments. Whereas ZOIDFS can describe noncontiguous I/O in a single function call, POSIX-based ROMIO drivers have to take other less efficient approaches, such as data sieving [27]. Moreover, the ZOIDFS driver can implement scalable file metadata operations such as `create`, `open`, `resize`, and `sync`. One client process executes the metadata operation and then broadcasts the results to the other clients [28]. On the Blue Gene/P, all processes in the communicator invoke the metadata system calls, thus resulting in increased file system traffic.

C. Buffered Message Interface

Buffered Message Interface (BMI) is a network abstraction layer designed for high-performance parallel I/O [21]. BMI enables parallel file systems to operate on multiple interconnection networks such as TCP/IP, InfiniBand, and Myrinet. While message-passing architectures such as Portals [29] and MPI [30] also provide network abstractions, BMI has inherent support for parallel I/O communication patterns. For instance, the BMI *list* operations allow applications to send or receive a set of noncontiguous buffers in a single function call.

BMI exports two sets of APIs: a user-level API and an internal device API. The user-level API is used by higher-level services such as file systems, whereas the device API is used for specific network implementations. The dual-layered

architecture enables BMI to abstract the details of the network from applications while exploiting the high-performance capabilities of modern interconnects. The BMI API is also asynchronous, thread-safe, and stateless. File systems can post and test for multiple I/O operations across several different networks simultaneously. This forms a basis for a portable, scalable, and concurrent communication paradigm.

BMI provides exactly the functionality that we need for the I/O forwarding framework. BMI was developed as part of the PVFS project and up until now, the implementation was only distributed with PVFS. We have removed the dependencies on PVFS so that BMI can exist as a standalone package. We have also fixed bugs uncovered as a result of using BMI in applications where it is used simultaneously for client and server communication in the same process.

D. ZOIDFS File System Interface

The ZOIDFS server is a daemon that runs on I/O nodes. It receives encoded I/O requests from compute nodes, decodes the requests, and performs I/O on behalf of the compute nodes. The current implementation uses a pool of threads to concurrently service requests from multiple clients.

Once the ZOIDFS server has received and decoded the I/O requests from the compute nodes, it invokes the corresponding file system driver via an I/O dispatcher. The dispatcher identifies file systems based on the ZOIDFS handle and calls the corresponding driver code. We have developed ZOIDFS drivers for PVFS and POSIX-compliant file systems.

It is fairly straightforward to map a stateless, handle-based file system API such as PVFS to the ZOIDFS protocol. The 8-byte PVFS handle can be incorporated in the 32-byte ZOIDFS handle, resulting in a one-to-one mapping between the ZOIDFS and PVFS API. However, mapping the stateless ZOIDFS I/O protocol with stateful POSIX-compliant file systems introduces significant challenges.

E. ZOIDFS on POSIX

Since POSIX is the only accessible client API for most file systems, we must find an efficient mapping between the ZOIDFS and POSIX APIs. Two issues complicate implementing the ZOIDFS API on top of a POSIX file system. First, since the ZOIDFS API does not require a client to indicate when it has finished using a file handle (i.e., a *close* operation), some form of garbage collection has to be implemented to free the resources associated with every open POSIX handle. Also, most operating systems limit the number of files an application can have open simultaneously.

The second issue arises when a client reuses a previously used handle. Since the mapping between a file and its associated ZOIDFS handle is immutable, an application can reuse a handle without first performing a lookup. When this situation occurs, the ZOIDFS POSIX driver needs to obtain a POSIX file handle for a given ZOIDFS handle. The problem is that, while most file systems internally employ a handle-like identifier to uniquely identify a file (for example, inodes), POSIX does not require them to expose this identifier to the user. Hence, using the POSIX interface, one can obtain a file handle only by specifying the full filename. In other words, the ZOIDFS POSIX driver needs to perform a "reverse lookup" (mapping a ZOIDFS handle back onto a filename) to reopen the file.

There are two ways to maintain a persistent mapping between ZOIDFS handles and filenames. In the first method, reverse lookups are implemented by using a database that stores $\langle$handle, filename$\rangle$ tuples. For each lookup, this database is consulted. If the filename is already present, its handle is returned. Otherwise, a unique ZOIDFS handle is generated, and the filename and handle are added to the database.

Unfortunately, this approach introduces a number of problems. For one, the size of the database is bounded by the number of files on the file system. Each file accessed through the ZOIDFS API will require an entry in the database, and – since handles are persistent – entries can be removed only when the file itself is removed. Hence, the database cannot be kept in memory.

In addition, to ensure the scalability of opening files in a parallel application, the ZOIDFS API explicitly allows for performing a single lookup on one process and broadcasting the resulting handle to other processes. Hence, the filename database needs to be shared by all processes using the ZOIDFS API. Although the fact that handles are immutable enables aggressive per-process caching, using a shared database does not offer a scalable solution.

The other option is to move the responsibility of providing the full filename to the application. This approach eliminates the disadvantages of maintaining a handle database. We have added a new error code, *ESTALE*, to the ZOIDFS API. If the ZOIDFS layer needs to obtain the filename associated with a ZOIDFS handle and is unable to do so (for example, because the underlying file system does not support it), it will return *ESTALE*. This indicates to the user application that it needs to perform a lookup operation on the file, thus re-establishing the $\langle$handle, filename$\rangle$ mapping.

Our current implementation of the ZOIDFS POSIX driver uses a least-recently-used (LRU) policy to limit the number of concurrent POSIX file handles. In addition, the driver can be configured to use a local, global or memory-only database to keep track of ZOIDFS handles.

V. Experiments

In this section we evaluate the performance of the I/O forwarding framework. We present results from metadata microbenchmarks, I/O benchmarks, and an application. We have used PVFS as the underlying file system in our experiments.

It is important to note the functional differences between IOFSL, which is an I/O forwarding framework, and PVFS, which is a parallel file system. An I/O forwarding framework initiates parallel file system operations. IOFSL encodes the function parameters at the compute nodes, sends the encoded parameters to the I/O node, decodes the function parameters at the I/O node, and then hands off the I/O operations to the file system. As such, in small-scale testing environments, IOFSL

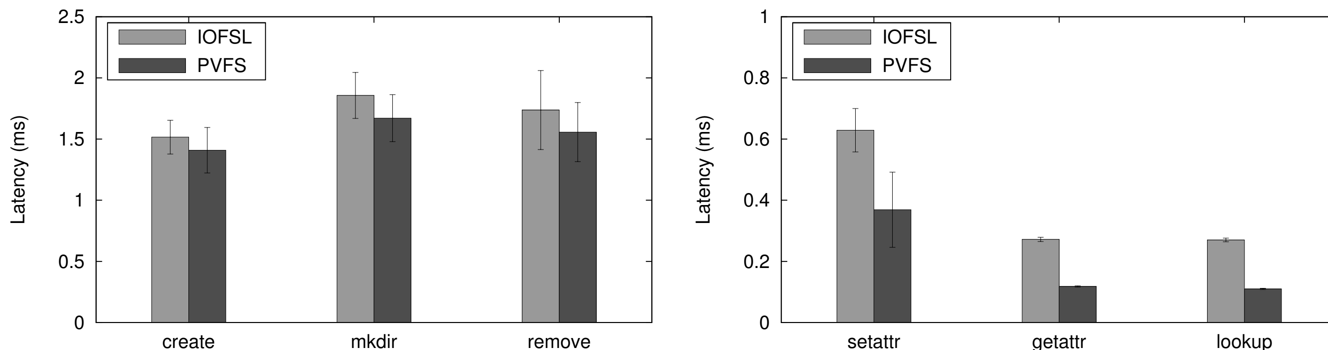


Fig. 5: Metadata operations latency. Left: create, mkdir, remove; right: setattr, getattr, lookup.

will always perform slower than direct access to the parallel file system. The potential benefits of I/O forwarding are realized primarily in large-scale, massively parallel computing environments where significant aggregation is possible. However, comparing the I/O forwarding framework to a parallel file system enables us to quantify the overhead associated with forwarding I/O system calls from the compute nodes to the I/O node.

The experiments were conducted on a Linux cluster. Each cluster node consists of dual AMD Opteron 250 processors, 2 GB of RAM, an onboard Tigon 3 Gigabit Ethernet NIC, and an 80 GB SATA disk. The nodes are connected via an SMC 8648T 48-port switch. The testbed consisted of a single PVFS server running a development version of pvfs-2.8.1, an I/O forwarding server, and compute node clients, all running on separate nodes. We bypass the libsysio layer to access the PVFS file system directly, by using a PVFS driver for the ZOIDFS API.

A. *Latency Microbenchmarks*

The first set of experiments measures the latency of some common metadata operations. Figure 5 shows the time taken by a single client to create and remove files, create directories, perform file lookups, and set and retrieve file attributes.

The IOFSL metadata latency is 0.2 ms to 0.3 ms more than that of PVFS for all metadata operations. This represents the fixed cost associated with encoding the function parameters at the compute node, network communication overhead, and decoding the parameters at the I/O node. It takes only about 0.30 µs to encode and decode a typical ZOIDFS data structure (`zoidfs_attr_t`). Thus, most of the overhead associated with I/O forwarding is a result of the communication costs between the compute and the I/O nodes. While this fixed cost is barely noticeable when we create and remove files and directories, it has a higher relative impact on operations with low latencies, such as file lookups and setting and retrieving of file attributes. However, this additional cost can be offset by aggregating metadata operations at the I/O node.

B. *ROMIO perf*

The ROMIO perf benchmark is an MPI-IO application that measures the I/O bandwidth of file systems. Each process writes a data array to a fixed location in a shared file using noncollective I/O and individual file pointers. The data is then read back to calculate the aggregate I/O bandwidth. ROMIO perf reports two sets of I/O bandwidth results: with and without data being flushed to the disk.

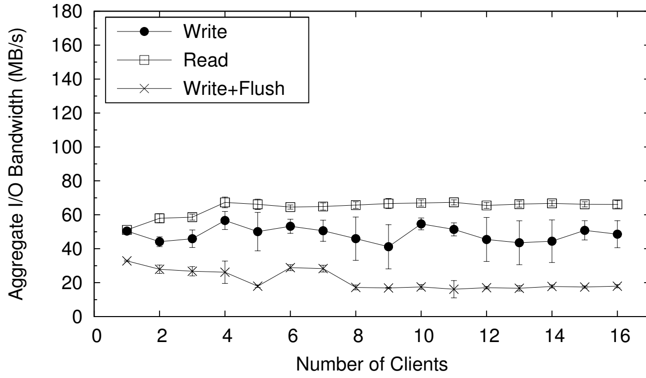
Figure 6 shows the aggregate I/O bandwidth of IOFSL and PVFS as a function of the number of clients. The read and write curves plateau almost immediately, signifying that only a few clients are needed to saturate the network. The IOFSL I/O bandwidth is lower than that of PVFS because of the costs associated with encoding and decoding the function parameters, communication overhead, and lack of I/O pipelining between the compute nodes and I/O nodes. The average IOFSL read bandwidth is about 40% less than that of PVFS while the average write bandwidth is about 16% less than that of PVFS. The write bandwidth with flushing enabled reflects the limit of the single SATA disk used in this experiment.

To study the benchmark without the limitations imposed by the disk throughput, we ran the experiment again after mounting the PVFS file system on a ramdisk. Figure 7 shows the new aggregate I/O bandwidth results. In the case of direct PVFS, the read bandwidth is again bound by the network. For both IOFSL and direct PVFS, the write bandwidth is comparable to the case where the file system is mounted on the disk, signifying that the ROMIO perf data is small enough to be cached in memory. Also, flushing the data after the write operation does not adversely affect the write bandwidth, as this operation is no longer limited by the disk throughput.

C. *NAS BTIO*

The BT benchmark is part of the NAS Parallel Benchmarks suite of applications. It solves systems of block-tridiagonal equations in parallel. BTIO [31] extends the BT benchmark by adding support for periodic solution checkpointing using noncontiguous MPI-IO calls. We used the Class C *full* version of BTIO, which uses collective I/O to generate large, regular I/O requests. BTIO requires that the number of clients be squares of integers. The Class C version of the benchmark is data-intensive, reading and writing almost 7 GB of data.

Figure 8 measures the BTIO Class C I/O bandwidth as a function of the number of clients. The BTIO write bandwidth



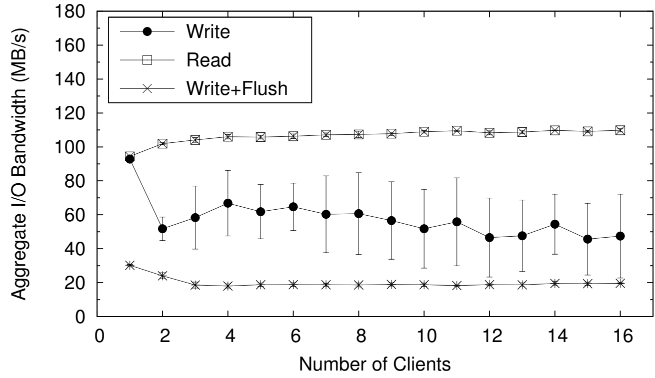
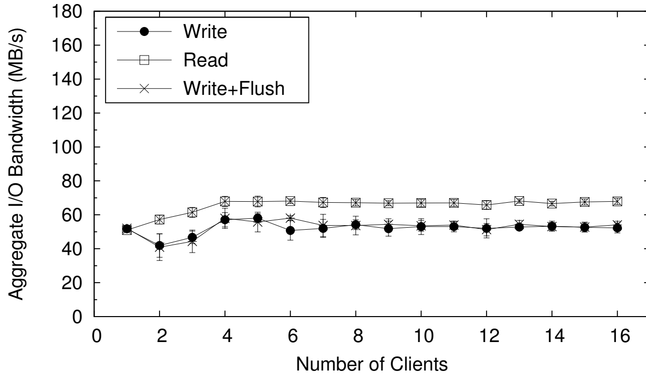


Fig. 6: ROMIO perf. Left: IOFSL/PVFS; right: direct PVFS.



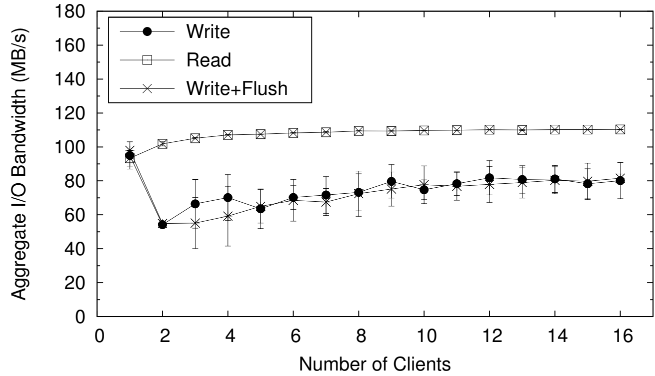


Fig. 7: ROMIO perf. Left: IOFSL/PVFS; right: direct PVFS. The file system is mounted on a ramdisk.

plateaus at about 20 MBps for both IOFSL and PVFS. This is primarily a limitation of the disk bandwidth of the single SATA disk used in all the experiments. The read bandwidth plateaus at about 35 MBps for PVFS and at about 30 MBps for IOFSL because of the limited available disk bandwidth. The difference between the PVFS and IOFSL read throughput can be attributed to the additional store-and-forward latency associated with moving the data from the compute nodes to the I/O node in the case of IOFSL.

The I/O bandwidth reported by ROMIO perf (Figure 6) is more than that of NAS BTIO for both IOFSL and direct PVFS. This result can be explained by the amount of data read and written by these benchmarks. Each perf client reads and writes 8 MB of data, which is small enough to be cached in memory. The BTIO Class C benchmark reads and writes almost 7 GB of data. Since this data is too large to be cached in memory, the BTIO I/O bandwidth is limited by the disk bandwidth. We note that in the case of perf, the write bandwidth is the same as BTIO when the data is flushed to the disk after each write operation.

D. Scalable Synthetic Compact Application

Scalable Synthetic Compact Application (SSCA) [32] is a set of high-performance computing benchmarks that model scientific applications such as bioinformatics optimal pattern matching, graph analysis, SAR sensor processing, and knowledge formation. We used the I/O-only version of the SSCA-3 code for these experiments.

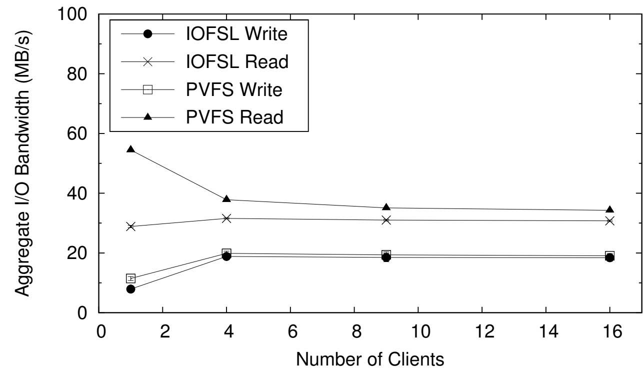


Fig. 8: BTIO Class C I/O bandwidth.

We made two modifications to the SSCA-3 code. First, we replaced the POSIX file I/O system calls with MPI-IO calls. This enables us to measure the PVFS I/O performance without the overhead associated with tunneling I/O requests through the PVFS kernel module. The second modification involved removing a behavior in the SSCA-3 code wherein the application would break the I/O operations into 4-byte chunks; that is, while the application generates large read and write requests, a subroutine breaks the requests into smaller chunks.

Figure 9 shows the normalized execution time of the SSCA-3 application for the three predefined test runs: *test0*, *test1*, and *test7GB*. The I/O and metadata footprint of the application progressively increases as we move from *test0* to *test7GB*. Table I lists the I/O and metadata information for the SSCA-3 benchmark.

TABLE I: SSCA-3 I/O and metadata footprint.

TEST	FILES	READ	WRITE	TOTAL
test0	498	0.35 GB	0.17 GB	0.52 GB
test1	2,910	2.17 GB	1.03 GB	3.20 GB
test7GB	92,634	18.90 GB	7.19 GB	26.09 GB

The SSCA-3 execution time for IOFSL is 21%–26% more than that of PVFS. This overhead is predominantly due to the additional store-and-forward latency associated with moving file data from the compute nodes to the I/O node. However, by aggregating I/O and metadata operations at the I/O node, we can improve IOFSL performance by reducing the file system traffic. We will address this issue in future work.

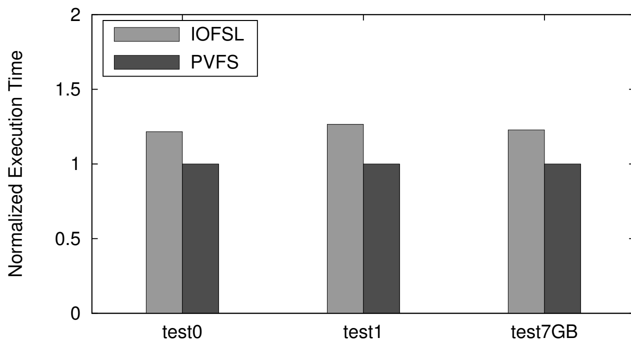


Fig. 9: SSCA-3 normalized execution time.

VI. RELATED WORK

Remote Procedure Call (RPC) is a communication mechanism that enables applications to execute the called procedure on a different host machine [33]. RPCs encode the function parameters that are then passed over the network to a remote server. The server executes the function call on behalf of the client and sends the results back to the client. I/O forwarding is essentially a specialized form of RPC, where the I/O function calls are sent to the I/O node for execution.

The Computational Plant (Cplant) [34] machine at Sandia National Laboratories introduced the concept of I/O forwarding in HPC systems. The Cplant compute nodes forward the I/O requests to a parallel job launcher called *yod*, which then performs I/O on behalf of the compute nodes. The IBM Blue Gene series of supercomputers use I/O forwarding to ship I/O operations from compute nodes to dedicated I/O nodes [11]. The Blue Gene compute nodes and I/O nodes are organized into multiple *processing sets (psets)*. Each *pset* consists of a single I/O node and a fixed number of compute nodes. I/O operations from the compute nodes are shipped to the corresponding I/O node over a collective network. A dedicated control daemon running on the I/O node performs I/O on behalf of the compute nodes.

A related research project at Argonne National Laboratory seeks to mitigate some of the design limitations of the Blue Gene I/O forwarding framework [4]. ZOID is an open and scalable I/O forwarding architecture for the IBM Blue Gene/P system. It defines a new I/O forwarding protocol for shipping I/O operations from compute nodes to I/O nodes. However, the ZOID I/O forwarding architecture is tightly coupled to the IBM tree network. It was designed for the Blue Gene series of supercomputers and is not portable to other HPC systems such as the Cray XT5 or Linux clusters.

VII. CONCLUSIONS

The performance mismatch between the computing and I/O components of the current generation of HPC systems has made I/O the critical bottleneck for data-intensive scientific applications. I/O forwarding attempts to bridge this increasing performance gap by regulating the file system I/O traffic. In this paper, we present an open, scalable I/O forwarding framework for high-performance computing systems. We document the potential benefits of I/O forwarding and quantify the overhead associated with introducing another layer in the I/O stack.

The I/O forwarding layer provides a platform for optimizing the file system I/O traffic. We plan to pipeline the I/O requests between the compute nodes and I/O nodes and leverage the knowledge of larger I/O patterns to aggregate I/O requests and perform data and metadata caching. The other areas for future work include designing a capability-based security model for the ZOIDFS protocol and integrating the SYSIO library with the I/O forwarding stack. We are also developing a BMI driver for the IBM Blue Gene/P tree network. This will enable the IOFSL code to run on the Intrepid system at Argonne National Laboratory. We will make the IOFSL source code available for download by the time of publication.

REFERENCES

[1] "Argonne Leadership Computing Facility," http://www.alcf.anl.gov.

[2] "Jaguar," http://www.nccs.gov/jaguar.

[3] Lawrence Livermore National Laboratory, "ASC Purple," https://asc.llnl.gov/computing_resources/purple.

[4] K. Iskra, J. W. Romein, K. Yoshii, and P. Beckman, "ZOID: I/O-forwarding infrastructure for petascale architectures," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. NY: ACM, 2008, pp. 153–162.

[5] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur, "PVFS: A parallel file system for Linux clusters," in *Proceedings of the 4th Annual Linux Showcase and Conference*, 2000, pp. 317–327.

[6] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*. Berkeley, CA: USENIX Association, 2002.

[7] Cluster File Systems, Inc., "Lustre: A scalable high-performance file system," Cluster File Systems, Tech. Rep., Nov. 2002, http://www.lustre.org/docs/whitepaper.pdf.

[8] D. Nagle, D. Serenyi, and A. Matthews, "The Panasas ActiveScale storage cluster—delivering scalable high bandwidth storage," in *Proceedings of the ACM/IEEE Conference on Supercomputing*, Pittsburgh, PA, Nov. 2004.
[9] Argonne National Laboratory, "ROMIO: A high-performance, portable MPI-IO implementation," http://www.mcs.anl.gov/romio.
[10] "Parallel-NetCDF," http://www.mcs.anl.gov/parallel-netcdf.
[11] 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 *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, February 2006.
[12] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan, "Operating system issues for petascale systems," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 2, pp. 29–33, 2006.
[13] P. Beckman, K. Iskra, K. Yoshii, S. Coghlan, and A. Nataraj, "Benchmarking the effects of operating system interference on extreme-scale parallel machines," *Cluster Computing*, vol. 11, no. 1, pp. 3–16, 2008.
[14] S. M. Kelly and R. Brightwell, "Software architecture of the light weight kernel, Catamount," in *Proceedings of the 47th Cray User Group Conference*, Albuquerque, NM, May 2005.
[15] J. E. Moreira *et al.*, "Designing a highly-scalable operating system: The Blue Gene/L story," in *Proceedings of the ACM/IEEE Conference on Supercomputing*, Tampa, FL, Nov. 2006.
[16] D. Wallace, "Compute node linux: New frontiers in compute node operating systems," in *Proceedings of the Cray User's Group*, 2007.
[17] IBM, "Overview of the IBM Blue Gene/P project," *IBM Journal of Research and Development*, vol. 52, no. 1/2, pp. 199–220, 2008.
[18] "FUSE: Filesystem in userspace," http://fuse.sourceforge.net/.
[19] "SYSIO," http://sourceforge.net/projects/libsysio.
[20] M. Eisler, "XDR: External data representation standard," http://www.ietf.org/rfc/rfc4506.txt.
[21] P. H. Carns, W. B. Ligon III, R. Ross, and P. Wyckoff, "BMI: A network abstraction layer for parallel I/O," in *Proceedings of IPDPS'05, CAC workshop*, Denver, CO, Apr. 2005.
[22] *InfiniBand Architecture Specification*, http://www.infinibandta.org/specs/, InfiniBand Trade Association, Oct. 2004.
[23] W. Liao and A. Choudhary, "Dynamically adapting file domain partitioning methods for collective I/O based on underlying parallel file system locking protocols," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. IEEE Press Piscataway, NJ, 2008.
[24] M. Fahey, J. Larkin, and J. Adams, "I/O performance on a massively parallel Cray XT3/XT4," in *International Symposium on Parallel and Distributed Processing*, April 2008, pp. 1–12.
[25] IEEE, *1003.1-1988 INT/1992 Edition, IEEE Standard Interpretations of IEEE Standard Portable Operating System Interface for Computer Environments (IEEE Std 1003.1-1988)*. NY: IEEE, 1988.
[26] R. Thakur, W. Gropp, and E. Lusk, "On implementing MPI-IO portably and with high performance," in *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*. NY: ACM Press, 1999, pp. 23–32.
[27] A. Ching, A. Choudhary, K. Coloma, W. K. Liao, R. Ross, and W. Gropp, "Noncontiguous access through MPI-IO," in *Proceedings of the IEEE/ACM International Symposium on Cluster Computing and the Grid*, May 2003.
[28] R. Latham, R. Ross, and R. Thakur, "The impact of file systems on MPI-IO scalability," *Lecture Notes in Computer Science*, vol. 3241, pp. 87–96, September 2004.
[29] R. Brightwell, B. Lawry, A. B. MacCabe, and R. Riesen, "Portals 3.0: Protocol building blocks for low overhead communication," in *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, 2002.
[30] MPI Forum, "MPI-2: Extensions to the Message-Passing Interface," http://www.mpi-forum.org/docs/docs.html, 1997.
[31] P. Wong and R. der Wijngaart, "NAS parallel benchmarks I/O version 2.4," NASA Ames Research Center, Moffet Field, CA, Tech. Rep. NAS-03-002, Jan. 2003.
[32] HPCS, "Scalable Synthetic Compact Application," http://www.highproductivity.org/SSCABmks.htm.
[33] R. Srinivasan, "RPC: Remote procedure call protocol specification version 2," http://www.ietf.org/rfc/rfc1831.txt.
[34] Sandia National Laboratories, "Computational Plant," http://www.cs.sandia.gov/cplant.

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.