

# Scalable I/O Forwarding Framework for Petascale Architectures

Nawab Ali\*, Phil Carns<sup>†</sup>, Kamil Iskra<sup>†</sup>, Dries Kimpe<sup>‡</sup>, Sam Lang<sup>†</sup>, Robert Latham<sup>†</sup>, Rob Ross<sup>†</sup>,  
Lee Ward<sup>§</sup>, P. Sadayappan\*

\*The Ohio State University

Email: {alin, saday}@cse.ohio-state.edu

<sup>†</sup>Argonne National Laboratory

Email: {carns, iskra, slang, robl, rross}@mcs.anl.gov

<sup>‡</sup>University of Chicago

Email: dkimpe@mcs.anl.gov

<sup>§</sup>Sandia National Laboratories

Email: lee@sandia.gov

**Abstract**—Current leadership-class machines suffer from a significant imbalance between their increasing computational power and the limited I/O bandwidth. While Moore’s law ensures that the computational power of high-performance computing (HPC) systems increases with every generation, the same is not true for their I/O subsystems. The limited scalability of existing parallel file systems, coupled with the minimalistic compute node kernels running on these machines, calls 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 gap between the compute and I/O components of HPC systems by shipping I/O calls from compute nodes to dedicated I/O nodes. The I/O nodes perform I/O 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 capable of running on massively parallel HPC systems such as the IBM BG/P, Cray XT5, and Linux clusters. We also 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] or the *Roadrunner* machine at the Los Alamos 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 by an addition in the number of nodes, along with 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. 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 Gflop, the I/O bandwidth-to-flops ratio of current leadership-class machines is around 100 GBps for 1 Pflop. This significant decrease in scale in terms of the bytes-to-flops ratio adversely affects the performance of data-intensive high-end computing applications, which often do not see a corresponding performance improvement on faster supercomputers. For leadership-class machines, I/O is the critical performance bottleneck.

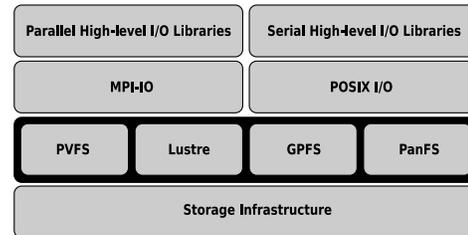


Fig. 1: Typical I/O software stack for HPC systems.

In view of the limitations imposed by current storage 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, file system implementations, and the storage infrastructure. The 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?

The parallel file systems available on current leadership-class machines, such as PVFS2 [3], GPFS [4], Lustre [5], and PanFS [6] were designed with smaller systems in mind. While some of these file systems incorporate features for enhanced scalability, they are essentially hamstrung by the limited throughput available from storage devices. Moreover,

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.

since not all HPC systems use the same parallel file system, attempting to address this challenge at the file system layer might prove ineffective. The other option is to make the scalability improvements at the MPI-IO layer. ROMIO [7] is the *de facto* standard MPI-IO implementation from Argonne National Laboratory. Since it is distributed as part of the MPI library, 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. Parallel high-level libraries such as Parallel-NetCDF [8] use MPI-IO and, as such, face many of the same limitations outlined above. 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.

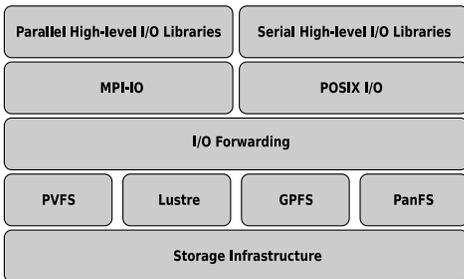


Fig. 2: I/O software stack with I/O forwarding.

Current HPC systems typically run a minimalistic operating system kernel on the compute nodes to limit the operating system (OS) “noise”. The IBM Blue Gene series of supercomputers also restrict I/O operations from the compute nodes to limit the I/O jitter from kernel-resident file systems. To enable applications to perform I/O, the compute node kernel forwards 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 bridging the application interface with the file system interface, the I/O forwarding layer has the potential to enhance the scalability and performance of I/O subsystems.

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 a single architecture [9], [10].

The main contributions of this paper are as follows:

- We present a generic I/O forwarding framework capable of running on massively parallel leadership-class machines such as the IBM BG/P, Cray XT5, and Linux clusters.
- We present a new protocol and API (ZOIDFS) for forwarding I/O function calls from the compute nodes to the I/O node.
- We quantify the overheads associated with introducing

the I/O forwarding layer in the I/O stack in relation to latency and bandwidth.

## II. I/O FORWARDING

General-purpose operating systems such as Linux are designed for a multiuser, 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 adversely affect the performance of computation-intensive HPC systems. The reason is that general-purpose OS kernels introduce significant levels of noise in the system in the form of context switches, cache poisoning, translation lookaside buffer (TLB) misses, and interrupts. These kernels are unable to scale to leadership-class machines because of the performance impact of the OS interference on HPC applications, particularly with respect to synchronicity [11], [12].

To mitigate the levels of noise in OS kernels, massively parallel machines such as the IBM Blue Gene/P and the Cray XT5 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 [13]. While limiting I/O support in the CNK may lead to better application performance, it also restricts the capabilities of parallel applications. In fact, the inability to perform file I/O may render most scientific applications useless. There are several ways to circumvent the I/O restrictions imposed by the CNK. Applications can use a user-level virtual file system (VFS) such as FUSE [14] or the SYSIO library [15] to perform file I/O. The SYSIO library provides POSIX-like file I/O support for remote file systems. A more interesting 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.

I/O forwarding bridges the increasing gap between the computational power of leadership-class machines and the limited scalability of parallel file systems. Current file systems are unable to service the concurrent requests from hundreds of thousands of processing elements. 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 number of clients accessing the file system. I/O forwarding also enables us to 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 allow the compute nodes direct access to the file systems, such as the Cray XT and Linux clusters.

### III. SCALABLE I/O FORWARDING FRAMEWORK

Figure 3 shows the I/O forwarding infrastructure on the IBM Blue Gene/P. The set of compute nodes is partitioned into smaller subsets and assigned to an I/O node. The typical 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 Gigabit Ethernet network. This hierarchical design enables the system to scale by limiting the number of clients that can directly access the file system. 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.

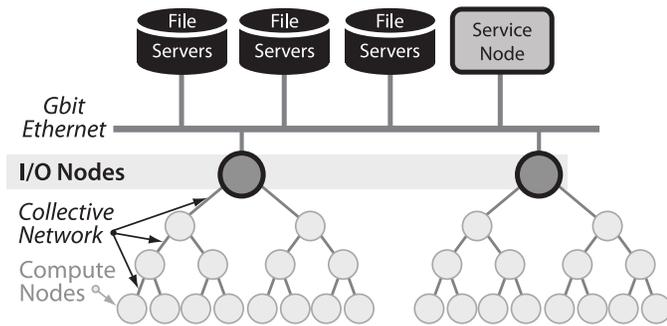


Fig. 3: I/O forwarding architecture for IBM BG/P.

There are some significant drawbacks associated with the Blue Gene/P I/O forwarding infrastructure. 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, thereby losing the optimizations performed at the MPI-IO layer, especially with respect to data sieving and I/O aggregation. Also, the BG/P I/O forwarding infrastructure is inflexible, proprietary, and tightly coupled to IBM technologies.

In view of the importance of I/O forwarding in the I/O stack of leadership-class machines and the suboptimal solutions available today, we propose a scalable, unified, high-performance computing I/O forwarding framework that will bridge the increasing performance gap between the computation power and I/O subsystems of petascale machines. 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 any or all parallel file system solutions.

- Support any or all high-speed interconnects and networking solutions.
- Integrate with MPI-IO and any hardware features designed to support efficient parallel I/O.

The I/O forwarding framework leverages the work done on the ZOID and ZOIDFS projects at Argonne National Laboratory [10]. In particular, we use the ZOIDFS I/O protocol and API as a starting point for our research. Figure 4 shows the software stack of the I/O forwarding scalability layer (IOFSL). 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 to keep the architecture generic. 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. We believe that a framework capable of running on multiple machine architectures, high-speed interconnects, and file systems has a higher chance of being adopted by the HPC community. In view of the above design requirement, we have used multiple levels of abstractions, at the client, network, and file system layers.

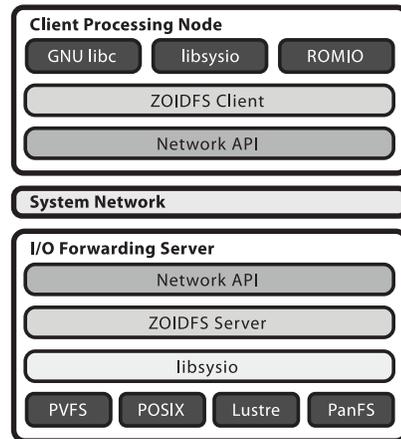


Fig. 4: I/O forwarding software stack.

The compute node component of the I/O forwarding framework supports multiple operating system kernels, including IBM Blue Gene/P CNK, Cray XT Compute Node Linux (CNL), and Linux. To account for possible heterogeneity between the compute node and I/O node architectures, we encode the function parameters using XDR [16]. Similarly, because of the abstraction provided by the SYSIO library, we support multiple file systems on the I/O nodes, including PVFS2, Lustre, UFS, and PanFS. The use of BMI [17] for communication enables the I/O forwarding framework to use several high-speed interconnects such as InfiniBand [18], Myrinet, and Gigabit Ethernet.

## IV. I/O FORWARDING SOFTWARE STACK

This section describes the individual components of the I/O forwarding software stack. We explain our design choices and discuss the tradeoffs.

### A. ZOIDFS I/O Protocol

The POSIX file I/O protocol inhibits the performance of file systems in the HPC domain. To avoid the associated performance overhead, stateless file systems such as PVFS2 do not maintain POSIX consistency semantics. Also, the POSIX API is not expressive enough to describe I/O patterns such as noncontiguous file access, which are often experienced in high-performance computing environments.

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, 32-byte file handles to describe the I/O operations. 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 flexible and more expressive than POSIX and requires fewer I/O calls for file operations. For instance, ZOIDFS has no file open or close calls. Applications perform a file lookup to obtain the file handle. All subsequent operations use the file handle to perform I/O.

The ZOIDFS lookup call accepts either a full pathname or a parent handle and a component name, and the data read/write calls can operate on multiple memory buffers and multiple regions of the file within a single call.

```
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 sub-operations 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. Typically, POSIX-compliant file systems match user credentials against file permissions during file open. Since the ZOIDFS protocol obviates the need for

opening and closing files, this authentication step is essentially bypassed. Further, since the file handles are opaque entities and can be exchanged freely among the client processes, the file authentication process can be easily circumvented.

We are developing a capability-based security model that will incorporate a capability field in every ZOIDFS function call. The ZOIDFS server will check a user's capability before performing any I/O operation. This approach should augment the existing security measures in current file systems.

### B. ZOIDFS Client Interface

Applications can ship I/O requests to the I/O forwarding server via multiple client interfaces. The ZOIDFS protocol provides a native client API for call forwarding. While applications can use native ZOIDFS calls to perform I/O, this API was designed primarily for use by higher-level libraries such as ROMIO [7]. An alternative would be to use userspace VFS implementations such as FUSE [14] or SYSIO [15] to redirect POSIX file I/O calls to the ZOIDFS API.

The FUSE kernel module enables us to transparently intercept POSIX file operations without requiring any application modifications. These operations are subsequently directed to the ZOIDFS library. Special care has to be taken to match the stateful API of FUSE with the stateless ZOIDFS API. We note, however, that although FUSE currently provides the most transparent and user-friendly option, it might not be the most efficient choice. Unlike the methods described above—that handle all I/O in userspace—I/O operations using FUSE first travel to the kernel, where they are redirected back to a userspace library. Further testing is needed to determine whether the overhead incurred by this userspace-to-kernel roundtrip is acceptable. For now, we expect SYSIO to be the primary means of redirecting POSIX file I/O calls to the ZOIDFS API.

The ROMIO driver for ZOIDFS enables parallel applications to perform I/O call forwarding via the MPI-IO [19] interface. It converts MPI file views and datatypes into offset-list pairs, thereby delivering good noncontiguous I/O performance. 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 ZOIDFS driver is currently available in the MPICH2 subversion trunk.

### C. Buffered Message Interface

The Buffered Message Interface (BMI) is a network abstraction layer designed for high-performance parallel I/O [17]. 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 [20] and MPI [21] also provide network abstractions, BMI has inherent support for parallel I/O communication patterns.

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 was developed for the PVFS2 parallel file system, and as such the BMI code is tightly coupled with the PVFS2 source code. However, since BMI also meets the communication requirements of the I/O forwarding framework, we decided to decouple BMI from PVFS2. Currently, BMI can be installed and used independent of PVFS2.

#### D. ZOIDFS Server

The ZOIDFS server is a daemon that runs on the I/O nodes. It receives the encoded I/O requests from the 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.

The real advantage of an intermediate server lies in the potential for performing optimizations at the I/O forwarding layer. The ZOIDFS server can leverage its knowledge of the global I/O pattern to potentially reduce file system traffic by aggregating I/O requests, reordering the I/O queue, performing I/O pipelining between the compute and I/O nodes and caching the data and metadata requests. While the existing server replays the I/O requests without any of the above optimizations, an enhanced server currently in development will incorporate them.

#### E. ZOIDFS File System Interface

Once the ZOIDFS server has received and decoded the I/O requests from the compute nodes, it invokes the corresponding file system calls. Since leadership-class machines often use different and incompatible parallel file systems (Lustre, PVFS2, GPFS, etc.), using the API of any specific file system would result in nonportable code. To achieve portability across multiple parallel file systems, we have used the SYSIO [15] library to provide a file system abstraction layer. The SYSIO library provides applications with a POSIX-compatible, user-space file I/O API. Its plug-in architecture allows easy integration with existing file systems.

Mapping the stateless ZOIDFS I/O protocol with stateful POSIX-compliant file systems introduces significant challenges. It is fairly straightforward to map a stateless, handle-based file system such as PVFS2 to the ZOIDFS protocol. The 8-byte PVFS2 handle can be incorporated in the 32-byte ZOIDFS handle, resulting in a one-to-one mapping between the ZOIDFS and PVFS2 API.

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 mapping 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.

Reverse lookups are implemented by using a database that stores  $\langle \text{handle}, \text{filename} \rangle$  tuples. For each lookup, this database is consulted. If the filename is already present, its handle is returned. If not, 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 assure 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.

By moving the responsibility of providing the full filename to the application, the disadvantage of maintaining a handle database can be avoided. A new error code, *ESTALE*, was added 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, re-establishing the  $\langle \text{handle}, \text{filename} \rangle$  mapping.

Our current implementation of the POSIX driver for ZOIDFS 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 (IOFSL). We present results from metadata microbenchmarks, I/O benchmarks, and an application.

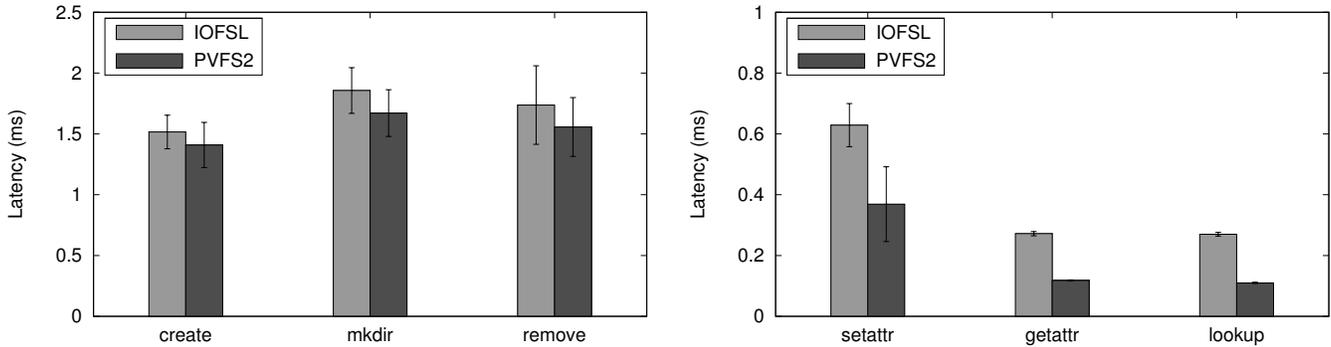


Fig. 5: Metadata operations latency; left: create, mkdir, remove; right: setattr, getattr, lookup.

We have compared the IOFSL framework with the PVFS2 parallel file system. It is important to note the functional differences between IOFSL, which is an I/O forwarding framework, and PVFS2, which is a parallel file system. A parallel file system performs only a subset of the operations of an I/O forwarding framework. 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, a parallel file system will always perform better than IOFSL. The potential benefits of I/O forwarding are realized primarily in large-scale, massively parallel computing environments where parallel file systems do not scale. However, benchmarking the I/O forwarding framework against 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 a 80 GB SATA disk. The nodes are connected via a SMC 8648T 48-port switch. The testbed consisted of a single PVFS2 server running a development version of pvfs-2.8.1, an I/O forwarding server, and compute node clients. We bypass the libsysio layer to access the PVFS2 file system directly, by using a PVFS2 driver for the ZOIDFS API. We hope to incorporate the libsysio layer in the IOFSL stack in the future.

#### A. Latency Microbenchmarks

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

The IOFSL metadata latency is 0.2 ms–0.3 ms more than that of PVFS2 for almost 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  $\mu$ s 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 significant impact on operations with low latencies, such as file lookups and setting and retrieving of file attributes.

#### B. ROMIO perf

The ROMIO perf benchmark is a 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 non-collective 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 PVFS2 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 PVFS2 because of the costs associated with encoding and decoding the function parameters and communication between the compute nodes and I/O nodes. 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 PVFS2 file system on the ramdisk. Figure 7 shows the new aggregate I/O bandwidth results. While the perf I/O bandwidth is still limited by the network, flushing the data to the disk no longer adversely affects the write bandwidth.

#### 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 [22] extends the BT benchmark by adding support for periodic solution checkpointing using noncontiguous MPI-IO calls. We used the *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

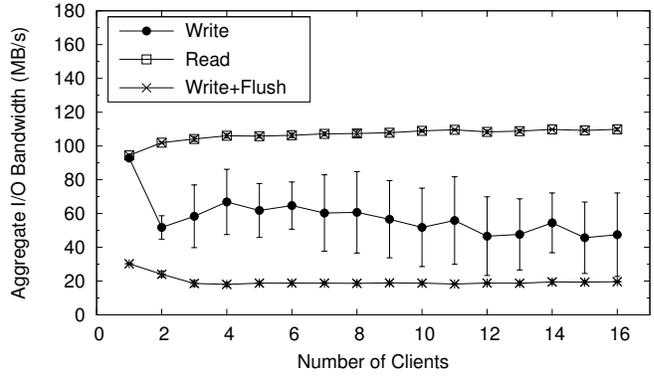
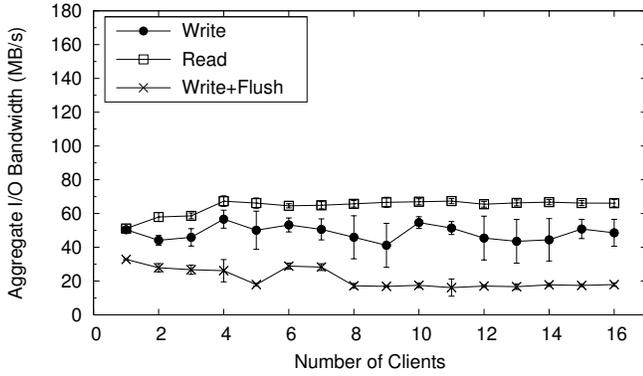


Fig. 6: ROMIO perf; left: IOFSL; right: PVFS2.

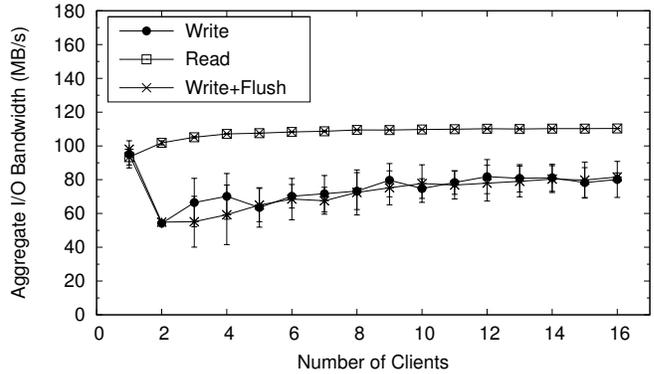
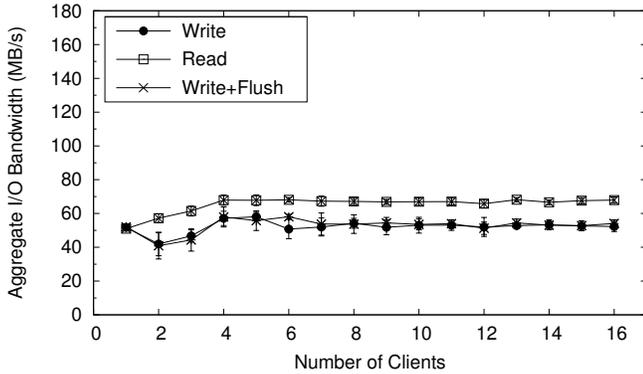


Fig. 7: ROMIO perf; left: IOFSL; right: PVFS2. The file system is mounted on a ramdisk.

plateaus at about 20 MBps for both IOFSL and PVFS2. 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 40 MBps for PVFS2 and at about 30 MBps for IOFSL because of the limited available network bandwidth. The difference between the PVFS2 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.

#### D. Scalable Synthetic Compact Application

Scalable Synthetic Compact Application (SSCA) [23] 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.

We made two modifications to the SSCA-3 code. First, we replaced the POSIX file I/O system calls with MPI-IO. This enables us to measure the PVFS2 I/O performance without the overhead associated with tunneling I/O requests through the PVFS2 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 kernel generates large read and write requests, a subroutine breaks the requests into smaller chunks.

Figure 9 shows the 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*. SSCA-3 *test7GB* creates about 100,000 files and reads and writes almost 7 GB of data.

The SSCA-3 execution time for IOFSL and PVFS2 is comparable for the *test0* and *test1* runs because of the small metadata and I/O footprints of these tests. For *test7GB*,

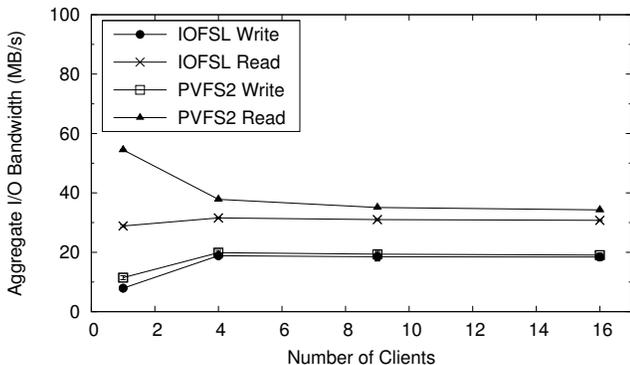


Fig. 8: BTIO Class C I/O bandwidth.

the PVFS2 execution time is almost 20% less than that of IOFSL. This overhead is predominantly due to the time spent in encoding and decoding the function parameters, and the additional store-and-forward latency associated with moving file data from the compute nodes to the I/O node.

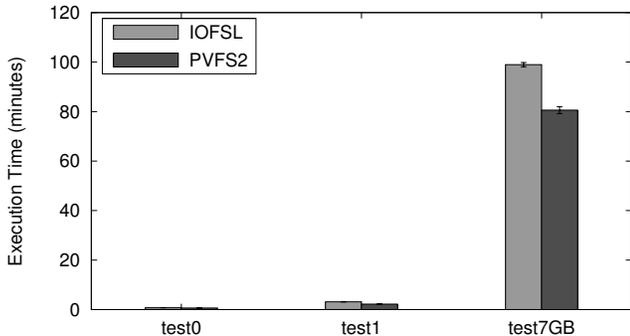


Fig. 9: SSCA3-IO execution time.

## VI. RELATED WORK

Remote Procedure Call (RPC) is a communication mechanism that enables applications to execute the called procedure on a different host machine [24]. RPCs encode the function parameters that are then passed over the network to a remote server. The remote 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) [25] 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 [9]. The Blue Gene compute nodes and I/O nodes are organized into multiple *processing sets* (*psets*). Each *psset* 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 console 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 [10]. ZOID is an open, scalable, and flexible 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. Section IV-A describes the ZOIDFS protocol in detail. 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, high-performance I/O forwarding framework capable of running on massively parallel HPC systems such as the IBM BG/P, Cray XT5, and Linux clusters. We document the performance 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 leverage the knowledge of global I/O patterns to implement I/O request aggregation, pipelining, and data and metadata caching. We are also working on a capability-based security model for the ZOIDFS I/O forwarding protocol.

## REFERENCES

- [1] "Argonne Leadership Computing Facility," <http://www.alcf.anl.gov>.
- [2] "Roadrunner," <http://www.lanl.gov/roadrunner>.
- [3] 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.
- [4] F. Schmuck and R. Haskin, "Gpfs: A shared-disk file system for large computing clusters," in *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*. Berkeley, CA: USENIX Association, 2002.
- [5] 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>.
- [6] D. Nagle, D. Serenyi, and A. Matthews, "The Panasas ActiveScale storage cluster—delivering scalable high bandwidth storage," in *Proceedings of the ACM/IEEE SC2004 Conference (SC'04)*, Pittsburgh, PA, Nov. 2004.
- [7] Argonne National Laboratory, "ROMIO: A High-Performance, Portable MPI-IO Implementation," <http://www.mcs.anl.gov/romio>.
- [8] "Parallel-NetCDF," <http://www.mcs.anl.gov/parallel-netcdf>.
- [9] 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 bluegene/l supercomputer," in *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA-12)*, February 2006.
- [10] 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: ACM, 2008, pp. 153–162.
- [11] 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.
- [12] 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.
- [13] IBM, "Overview of the IBM Blue Gene/P project," *IBM Journal of Research and Development*, vol. 52, no. 1/2, pp. 199–220, 2008.
- [14] "FUSE: Filesystem in userspace," <http://fuse.sourceforge.net/>.
- [15] "SYSIO," <http://sourceforge.net/projects/libsysio>.
- [16] M. Eisler, "XDR: External data representation standard," <http://www.ietf.org/rfc/rfc4506.txt>.
- [17] 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.

- [18] *InfiniBand Architecture Specification*, <http://www.infinibandta.org/specs/>, InfiniBand Trade Association, Oct. 2004.
- [19] 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*. New York, NY, USA: ACM Press, 1999, pp. 23–32.
- [20] 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.
- [21] MPI Forum, "MPI-2: Extensions to the Message-Passing Interface," <http://www.mpi-forum.org/docs/docs.html>, 1997.
- [22] 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.
- [23] HPCS, "Scalable Synthetic Compact Application," <http://www.highproductivity.org/SSCABmks.htm>.
- [24] R. Srinivasan, "RPC: Remote procedure call protocol specification version 2," <http://www.ietf.org/rfc/rfc1831.txt>.
- [25] 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.