ANL/MCS-TM-302

Extending the POSIX I/O Interface: A Parallel File System Perspective

by
M. Vilayannur[1], S. Lang[2], R. Ross[2], R. Klundt[3], and L. Ward[3]
[1]VMWare, Inc.
[2]Mathematics and Computer Science Division, Argonne National Laboratory
[3]Sandia National Laboratories

October 2008

UChicago ▶ Argonne LLC

Office of Science
U.S. DEPARTMENT OF ENERGY

A U.S. Department of Energy laboratory managed by UChicago Argonne, LLC

Contents

Extending the POSIX I/O Interface: A Parallel File System Perspective

Murali Vilayannur
VMWare, Inc.
muraliv@vmware.com

Samuel Lang, Robert Ross
Mathematics and Computer Science Division
Argonne National Laboratory
{slang,rross}@mcs.anl.gov

Ruth Klundt, Lee Ward
Sandia National Laboratories
{rklundt,lee}@sandia.gov

Abstract

The POSIX interface does not lend itself well to enabling good performance for high-end applications. Extensions are needed in the POSIX I/O interface so that high-concurrency HPC applications running on top of parallel file systems perform well. This paper presents the rationale, design, and evaluation of a reference implementation of a subset of the POSIX I/O interfaces on a widely used parallel file system (PVFS) on clusters. Experimental results on a set of micro-benchmarks confirm that the extensions to the POSIX interface greatly improve scalability and performance.

1 Introduction

POSIX [8] is the IEEE Portable Operating System Interface for Computing Environments and defines a standard interface for applications to obtain standard services from the operating system (such as networking, file system, memory management and process management). The design of the well-known POSIX I/O interfaces (`open`, `close`, `read`, `write` and `stat`) stems from the early 1970s and was intended primarily for a single machine with a single memory space accessing a simple peripheral device (such as a hard disk or tape). It is well understood that the semantics of these interfaces make it hard to achieve good performance when a large number of machines access shared storage in concert [12], a common pattern in parallel I/O applications of today. Many network file systems, such as NFS [19], do not adhere strictly to the POSIX I/O semantics to enable higher performance. Cluster file systems that adhere strictly to the semantics compromise on performance, and those that do not, compromise on correctness when workloads share files heavily.

In order to enable high performance for HPC/parallel I/O applications, a working group of HPC users [6] was constituted to propose and augment the POSIX API to provide standardized mechanisms and interfaces. The underlying philosophy of the

interfaces is to relax semantics that are expensive from a performance standpoint or enable passing higher-level information (such as access patterns) down to the file system and storage stack. Furthermore, any proposed interfaces must be simple for widespread adoption and still enable good performance for high-end applications.

Previous studies [20] have shown that commonly occurring patterns in high-end computing workloads are:

- concurrent file accesses
- noncontiguous file accesses
- high metadata rates (concurrent creates and deletes) and interactive use

In this paper, we provide the design and reference implementation of a subset of the POSIX I/O extensions for PVFS([3, 13]), a popularly used parallel file system on Linux clusters. The goal of these interfaces is to improve the performance of high-end applications for the kinds of access patterns outlined above. In particular, we group the implementation of the system calls under the following categories:

- Shared file descriptors/group opens (`openg, openfh`)
  An ever increasing number of nodes in large-scale computing environments necessitate a new approach to name-space traversal. One possible solution to reduce the overhead of concurrent name-space traversal is implementation of shared file descriptors at the file system. The prototypes for the system calls in this category look as follows:

```
int openg(const char *pathname,
          void *handle,
          size_t *handle_len,
          int flags,
          int mode);

int openfh(void *handle,
           size_t handle_len);
```

  The `openg` system call is expected to open/create a file specified by *pathname* according to the mode (permissions) and flags (read, write, truncate, etc.) specified. It fills in an opaque group handle in the buffer specified by *handle* and stores the size of the group handle in *handle_len*. The return value of this system call is 0 in the case of a successful call and -1 in the case of error (the error code is stored in the `errno` variable). The opaque group handle is intended to be passed to cooperative processes, which can then convert this into a file descriptor using the `openfh` system call to reference the same file system object with similar access rights. It is up to the implementation to limit the lifetime, scope, and security of the group handle.

  The `openfh` system call establishes an association between the group handle (that was returned from the `openg` system call) and a file descriptor. The file

descriptor that is returned from this system call (similar to the `open` system call) can be used to perform I/O (or used in any other system calls that require file descriptors). It is again left to the implementation to ensure that this system call does not incur unnecessary network overhead and that the opaque group handle contains sufficient information to facilitate conversion to an open file descriptor. The `openfh` system call interface can be envisioned as being equivalent to a distributed `dup` system call for a cluster environment that does not require any communication with the file servers.

- Lazy metadata attributes (`statlite, lstatlite, fstatlite`)
  This family of system calls ensures that I/O performance does not suffer because of needless attribute lookups (`stat`). These extensions are similar to the `stat` system call interface with the exception that they return the attributes wherein some values are not maintained at the same granularity as expected or not filled unless explicitly requested. For instance, the timestamps that keep track of the last access (*atime*) and update (*mtime*) may not be kept up to date on many cluster file systems because they tend to degrade performance. Another example of a lazy metadata attribute is the size of the file, which usually requires multiple network messages to be computed correctly and is not retrieved unless explicitly requested.

  ```
  int statlite(char *pathname, struct stat_lite *slbuf);

  int fstatlite(int fd, struct stat_lite *slbuf);

  int lstatlite(char *pathname, struct stat_lite *slbuf);
  ```

  The `stat_lite` structure is identical to the `stat` structure with the exception of a mask that specifies which attributes of the structure are requested and valid.

- Noncontiguous read/write interfaces (`readx, writex`)
  This family of system calls generalizes the file vector to memory vector data transfers. Existing vectored system calls (`readv, writev`) specify a memory vector (a list of offset, length pairs) and initiate I/O to contiguous portions of a file. The proposed system calls (`readx, writex`) read/write strided vectors of memory to/from strided offsets in files. The specified regions may be processed in any order. Although these system calls are similar to the *POSIX listio* interface in terms of reading/writing from noncontiguous regions of a file, they remove a number of shortcomings of the listio interface. The listio interface imposes a one-to-one correspondence between the sizes specified in the memory vector and the file regions and requires that the number of elements in the memory and file vector be the same. Furthermore, the `readx, writex` interface specifies that the implementation is free to do any reordering, aggregation or any other optimization to enable efficient I/O completion.

```
ssize_t readx(int fd,
              struct iovec *iov,
              size_t iov_count,
              struct xtvec *xtv,
              size_t xtv_count);

ssize_t writex(int fd,
               const struct iovec *iov,
               size_t iov_count,
               struct xtvec *xtv,
               size_t xtv_count);
```

The *xtvec* structure specifies the file offset (relative to the start of the file) and the number of bytes, and the *iovec* structure specifies the starting address of the memory buffer and the size of the buffer. The `readx` function reads *xtv_count* blocks described by the *xtv* structure from the file associated with the file descriptor into the *iov_count* buffers specified by the *iov* structure. Analogously, the `writex` functions writes the blocks described by the *iov* structure to the file at the offsets specified in the *xtv* structure. Since the underlying implementation is free to reorder the submitted requests, error semantics are a bit hard to define. The implementation described in this work returns the first error it notices (if any) or the number of bytes written (or read) successfully as the return value.

- Bulk metadata operations
  For many years, archiving and backup applications have lacked a portable bulk metadata interface to the file system. We provide a system call that returns file attributes with each of the directory entries read (a combination of `getdents` and `lstat` similar to the NFSv3 *readdirplus* request).

  ```
  int getdents_plus(int fd,
                    struct dirent_plus *dplus,
                    unsigned int count);
  ```

  This *dirent_plus* structure is similar to the *dirent* structure with the addition of fields for the attributes of the directory entry (*stat* structure) and an error code for the attribute operation (on failure).

- Hybrid metadata operation
  This system call is a hybrid of the lazy metadata system calls and bulk metadata operations described earlier. Rather than retrieving the familiar `stat` structure as part of the proposed getdents_plus system call interface, this system call allows retrieval of a subset of an objects' attributes.

  ```
  int getdents_plus_lite(int fd,
                         unsigned long lite_mask,
                         struct dirent_plus_lite *dpluslite,
                         unsigned int count);
  ```

The *lite_mask* argument dictates which attributes the caller is interested in and the *dirent_plus_lite* structure is similar to the *dirent* structure with the addition of fields for describing all/subset of valid attributes of the file objects (`stat_lite` structure) and an error code for the attribute operation (on failure).

The rest of this paper is organized as follows. Section 2 provides an overview of parallel file systems, with emphasis on the PVFS file system design. Implementation details of the POSIX system call extensions on top of PVFS are provided in Section 3, and experimental results are presented in Section 4. Related work is summarized in Section 5. Section 6 concludes with the contributions of this paper and discusses directions for future work.

2 Background

Parallel I/O continues to be an important aspect for enabling high-performance of computational science applications. Parallel file systems enable I/O-bound applications to scale by striping file data [3, 13] across multiple nodes of a cluster. PVFS2 is one such open-source parallel file system for Linux clusters that utilizes commodity networking and storage hardware to enable scalable, high-bandwidth I/O.

PVFS2 has a single server process running on a set of nodes in the cluster (see Figure 1). Each server process assumes a role (that of a meta server, data server, or both) that is indicated by the configuration files. The clients and servers are designed to handle numerous concurrently running operations, and a non blocking event-driven design built atop a state machine architecture allows these components to scale well with an increasing number of simultaneous operations.

PVFS2 supports multiple interfaces, including an MPI-I/O [16] interface via ROMIO [22] as well as the well-known POSIX interfaces by means of a kernel module and an associated user-space daemon that allow existing UNIX I/O utilities and programs to work on PVFS2 files without being recompiled. The design of the kernel module is similar to the *Coda* [11] implementation that queues VFS operations (and/or data) to a device file, which is then marshaled by a user-space daemon to the servers and returns responses back to the device (see Figure 2). In the case of I/O operations (such as `read` or `write`) that may transfer large amounts of data, there is a need to minimize the number of data copies and context switches [23]. Consequently, staging the data copies through intermediate kernel buffers is not an option. Instead, the kernel module orchestrates a user-space to user-space copying either before the write is initiated (from the I/O application to the client-side daemon buffers) or after the read operation is complete (from the client-side daemon to the I/O application buffers) to ameliorate the copying overheads. Alternatively, the client daemon could `mmap` the application buffers into its address space to avoid even the single copying overheads.

PVFS2 allows for native support of noncontiguous access patterns, which are found in many scientific applications (such as the Flash astrophysics application [1, 5]). PVFS2 user programs construct an efficient data structure using a set of routines (provided in the library) that represents a set of noncontiguous data regions that are to be read from or written to both on file and in memory. These data structures (after

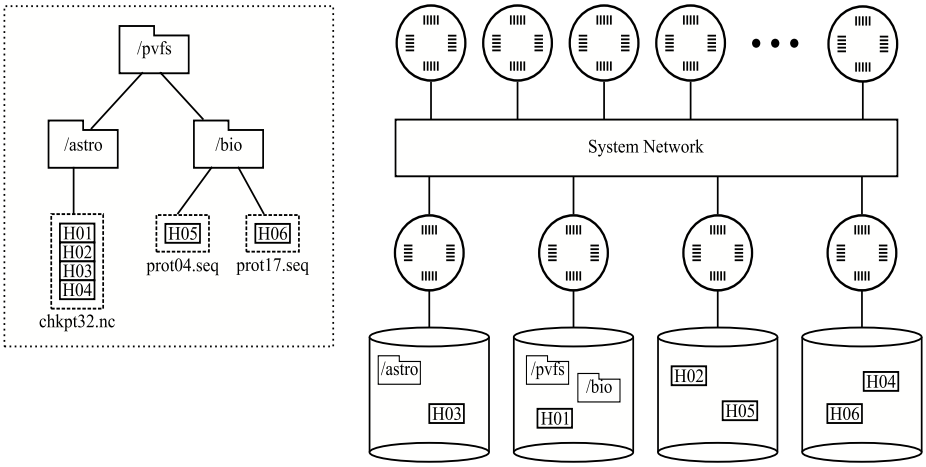

Figure 1: PVFS2 system architecture, shows the overall architecture of how the name-space is stitched together and the striping of file data across multiple I/O servers on a cluster

appropriate flattening and encoding) can then be transported to the servers to communicate the list of regions on which to perform I/O operations. We exploit this feature of PVFS2 to implement the noncontiguous POSIX (*readx/writex*) extensions efficiently as described in the next section.

The process of initiating access to files on PVFS2 is similar to the NFSv3 model whereby file names are translated (by means of a *lookup* operation) to an opaque 96-bit value (32-bit fsid and a 64-bit handle) that uniquely identifies an object (files, directories, and symbolic links) in the file system hierarchy. Given a handle to a file, any client program can use it to access regions of the file, read the contents of the directory or follow links as appropriate. At the time of system setup, PVFS2 servers are assigned handle ranges that enable clients to determine locations of servers handling a particular file system object/handle. New objects in the file system name-space are created by randomly choosing a server (to evenly distribute metadata storage responsibility to all servers) and requiring that the server create an object from its handle ranges.

Regular files have a layer of abstraction to deal with the striping of file data across different servers. As part of the metadata of the file object, an array of handle numbers for each stripe of the file is stored (for clarity, we will refer to the latter as *dfile handles* since they store the handles of the data files, and the former as *metafile handles* since it stores the metadata). This strategy is similar in spirit to the continuation inodes described in [7].

File handles are not special and do not have any associated lifetimes (other than the natural lifetime of the file system object to which they are assigned) or security context. Consequently, a file previously referenced (by means of a *lookup* operation) can be passed to other processes (using sockets, MPI messages, etc.), allowing them to access the same file system object. The PVFS2 ROMIO interface uses this technique to implement the `MPI_File_open` call with a single *lookup* to the file system, followed

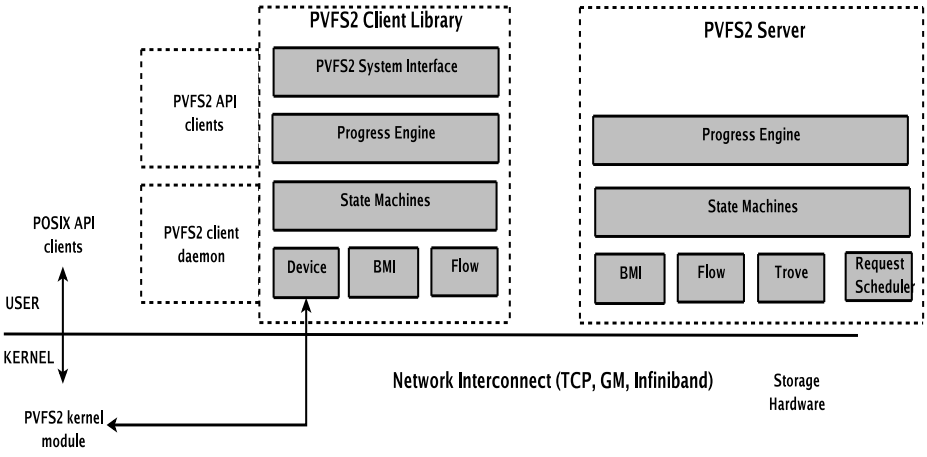

Figure 2: Depiction of the various components that make up the PVFS2 system software stack.

by a broadcast. We leverage this technique to implement the *group open* POSIX extensions by means of shared file descriptors in PVFS2, as elaborated in the next section.

3 Implementation

First, we describe the Linux VFS objects and terminologies in Section 3.1, before outlining the implementation details for shared file descriptors in Section 3.2. Noncontiguous read/write I/O interfaces are outlined in Section 3.3, lazy metadata attribute retrieval in Section 3.4 and bulk metadata operations with an emphasis on directory reading in Section 3.5.

3.1 Linux VFS: Objects and terminology

There are 4 critical VFS objects in the Linux kernel, namely the *superblock*, *file*, *inode* and *directory entry*. The superblock stores summary information of an entire mounted file system instance (such as used space, free space, file system identifiers and locations of root inode objects). Operations that can be performed on the super block typically include unmounting a volume, deleting an object, querying for the summary information and so on. The inode object is an in-memory representation of a physical file system object and encapsulates a file's data, metadata and extended attributes (such as owner, access times etc). Operations that manipulate the file system namespace such as `create, unlink and rename` are included in the inode operations. The file object represents an open instance of a file or directory object. Every file descriptor obtained from the `open, dup2, dup` system calls map to a file object. Operations that manipulate a file (such as opening, reading or writing) are grouped in the file operations. The directory entry object represents a cached name for an inode structure

in memory. On a *lookup*, directory entry objects for every component encountered in the path is created by the VFS. A directory entry object may point to only one inode object, but the converse may not be true, i.e. an inode object may have multiple directory entries pointing to it because of hard links (where different directory entries exist, but all of them point to the same file object). In addition, the VFS maintains a *dentry cache* (dcache) in addition to an *inode cache* (icache). Operations that are typically performed on a directory entry object include revalidating dentries, comparison of names and hashing names. For each of the POSIX interface extension described below, we outline the changes needed in the VFS layer using the above terminology.

3.2 Shared File Descriptors

File descriptors are process specific and their scope is limited to a process while group handles have namespace and scope global to the entire file system. To implement shared file descriptors, we need support from the underlying file system to

- ◇ encode an opaque group handle that can be shipped to remote machines (`openg`) and
- ◇ decode the opaque group handle to associate with an open file descriptor (`openfh`).

Note that the opaqueness of the group handle is with respect to user-space alone; the VFS and underlying file systems do impose implementation specific structure to the group handle. As noted earlier, it is left to the implementation to determine the lifetime and security context of the constructed group handle. Further, the group handle buffer needs to be encoded in order to maintain compatibility across heterogeneous architectures with different endian-ness. The current implementation defines a group handle to be composed of two components: a file system-independent portion (that is understood by the VFS) and a file system-specific opaque buffer (that is translated by individual file systems to obtain a file descriptor). The independent portion is encoded using a little-endian byte-first scheme.

The independent portion of the group handle comprises the *file system identifier (fsid)*, *flags* passed at the time of the `openg` system call, a *crc32 checksum*, and a *keyed SHA1 message authentication code*. Since the Linux VFS layer does not store the *fsid* as part of the per-file-system superblock, this needs to be queried for using the *statfs_lite* callback from the underlying file system. Since the `statfs` callback fills in many more fields than desired, the super operations structure was augmented with a *statfs_lite* callback that takes in a mask argument that determines the fields that the caller is interested in. This change is motivated by the fact that `statfs` on most cluster file systems can be quite expensive, since it must acquire read-only cluster locks on the file system structures before it can return the number of available or free blocks. The *flags* argument specified at the time of `openg` is encoded in the group handle so that clients that call `openfh` subsequently do not perform operations that they are not permitted to. The *checksum* and *HMAC* fields prevent any unauthorized tampering of the group handle by malicious clients. The key for the HMAC computation is provided by the underlying file system and is obtained by invoking a new callback to the super operations structure (*get_fs_key*). By encoding all these fields in the generic portion of the group handle, the VFS implementation on remote node kernels can verify the

authenticity of the group handle at the time of `openfh` before passing it down to the lower-level file system.

In PVFS2, the file system-specific portion of the group handle is constructed by simply encoding the PVFS2 handle of the file object (*fill_inode_handle* callback of the inode operations structure) in addition to the metadata attributes of the object (copied from the in-memory inode of the file). As stated earlier, since PVFS2 file handles can be freely shared among unrelated processes to access any object in the file system hierarchy, this implementation is fairly straightforward.

Upon receiving the group handle as part of the `openfh` system call, the VFS checks its authenticity by recomputing the *crc32* and *HMAC codes*. Any illegal tampering with the group handle is promptly signaled to the caller. Using the *fsid* stored in the group handle, the VFS locates the file system superblock structure corresponding to that (recall that *fsid* of the superblocks are obtained by calling the augmented *statfs_lite* callback). Upon successfully locating the superblock, the VFS calls into the lower-level file system with the file system-specific opaque buffer. This routine parses and verifies the authenticity of the opaque buffer and returns a pointer to an in-memory inode that represents the underlying file object. In PVFS2, this process involves extracting the object handle and using that to either locate the inode structure from the inode cache or allocating a new inode structure to represent the file object (*find_inode_handle* callback of the super operations structure). Since the purpose of shared file descriptors is to eliminate the use of file or path names, the inode structure is then associated with an anonymous directory entry and the first unused file descriptor for the calling process. An important thing to note in the PVFS2 implementation of `openfh` is that it does not require any communication with the server and is carried out completely locally. This is an important factor that contributes to the group open scalability as our results show in Section 4.

3.3 Noncontiguous Read/Write Interfaces

To implement the noncontiguous read/write interfaces efficiently, we need support from the underlying parallel file system to minimize the number of network messages and disk I/O. Since the `readx`, `writex` family of system calls generalizes the vectored I/O model from memory to file (noncontiguous in file as well as memory), a generic implementation could implement the desired functionality by issuing a sequence of `readv`/`writev` (or even `read`/`write`) calls. However, this approach would issue separate requests for each contiguous file region, rendering it increasingly inefficient as the number of regions grows. Our implementation augments the file operations structure with an optional callback to pass the entire vectored memory and file regions descriptions. This allows the underlying file system to decide the best way to orchestrate the I/O and network transfers.

PVFS2 is designed to handle noncontiguous I/O (both memory and file) efficiently. Similar to MPI data-types [16], PVFS2 provides mechanisms to construct efficient representations of noncontiguous data buffers and file regions. Once constructed, these structures (denoted as PVFS requests) can be passed to the I/O routines (and eventually to the servers). The effect is that PVFS2 allows us to collect the possibly noncontiguous regions, transport them to the appropriate servers (based on the file distribution scheme)

using as few messages as possible, and deposit them to the appropriate file offsets based on the file region structures that were exchanged earlier.

Similar to the `write` implementation, the `writex` implementation transfers the memory buffers described by the memory vector into consecutive buffers on the client daemon's address space, along with the associated file region vectors. Once the transfer is done, the client issues an I/O system interface call to initiate a write from the contiguous memory region to file by constructing succinct PVFS request descriptions to indicate the noncontiguity. The `readx` implementation transfers only the file region vectors at first, and the data transfer into the memory buffer happens after the I/O operation completes.

3.4 Lazy Metadata Attribute Retrieval

To implement lazy metadata attribute retrieval, we introduce new optional callback routines (*getattr_lite callback*) as part of the inode operations structure to indicate the attribute masks of the fields that need to be filled in. As mentioned earlier, some attributes of file system objects are expensive to maintain accurately (most notably access times), and some attributes are calculated only on demand (such as file sizes). Consequently, callers of these system calls explicitly indicate which attributes are desired, and it is left to the underlying implementation to optimize the algorithm used for fetching those attributes. In PVFS2, access and modification times are flushed lazily (usually at the time of an fsync (sync) or at the time of a file close), and the underlying implementation can ameliorate overhead by batching the time-stamp attribute changes. Consequently, a `stat` system call may not return the latest time stamps of a file system object by design because that process involves implementing a stateful locking subsystem. Since PVFS2, like other parallel file systems, stripes file data across a set of servers, maintaining file sizes on a single server (accurately) introduces a central point of bottleneck. Thus, PVFS2 computes file sizes on demand by communicating with all relevant data servers [3], a process that can be quite expensive. At this time, the only lazy attribute returned by PVFS2 as part of the `statlite` family of system calls is the size of the file, since the time-stamp attributes are lazy by design.

3.5 Bulk Metadata Operations

Parallel file systems are tuned for bulk data operations (bandwidth) typically, and do not expose interfaces that facilitate bulk metadata operations (latency). The lack of the latter can be critical (especially in the absence of client-side metadata caches or poor metadata cache hit rates) in terms of performance both for interactive tools such as the ubiquitous "ls" and for backup software [10] that traverses the entire (or subset of) namespace. Tools such as these typically read through all the entries of a directory tree and repeatedly call `stat` to retrieve attributes of each entry. Although the client-side name and attribute caches may filter a good number of accesses (*lookup* and *getattr* messages) from hitting the servers, the servers may still be overwhelmed enough that I/O performance is degraded significantly. Consequently, bulk metadata interfaces are needed to reduce overhead on the servers.

The proposed `getdents_plus` system call as described earlier not only returns a specified number of entries of a directory but also returns their attributes if possible (similar to NFSv3 *readdirplus*). It is entirely left to the underlying file system implementation to optimize the number of messages to retrieve the requested entries and their attributes. The file operations structure is augmented with a *readdirplus* and a *readdirpluslite* callback to facilitate efficient directory read operations at the underlying file system.

In PVFS2, both data and meta data are distributed over a set of servers. A call to read only the entries of a directory can usually be accomplished with a single message to a single server. However, the attributes of the directory entries may be located on different servers. Furthermore, for regular file objects, the sizes are usually computed by communicating with all the data servers. Therefore, a naive implementation could give rise to a flurry of network messages that could overwhelm all the servers. Consider a single directory with n directory entries (all of which are files), and a system composed of m servers. Assume that all servers can function as metadata and data servers. Assume that, on average each server holds the attributes of n/m file objects. For a naive implementation that does these operations sequentially, the total number of request-response message pairs would be $(1 + n * (m + 1))$, 1 for the initial directory read, n for retrieving the attributes of the objects, and $n * m$ for computing the file sizes of all the file objects.

We implement a smarter two-phase algorithm for accomplishing this operation efficiently in PVFS2. Specifically, we implement a vectored *listattr* request and state machines on the client and server. This request takes in a set of PVFS object handles as input and returns their attributes upon which the two-phase algorithm relies on. After the entries of the directory are read from the server, the first phase aggregates groups of file handles that are colocated on the same servers. Once aggregated, a set of *listattr* messages is sent to all the relevant servers (1 message pair per server) to fetch the attributes of the desired handles. If any of the handles refer to file objects (and file sizes are also desired), phase 2 begins by aggregating groups of data file handles that are colocated on the same servers. Once aggregated, a set of *listattr* messages is sent to all the relevant servers (1 message pair per server) to fetch the sizes of the data file handles. After phase 2 completes, the state machine collates the attributes and file sizes of all the objects that were obtained from the directory read. For the same parameters shown above, the total number of request-response message pairs is $1 + 2 * m$, 1 for the initial directory read, m for retrieving the attributes of the meta-handles, and another m for retrieving the sizes of the data-handles. Thus, the two-phase algorithm reduces the message complexity from O(m * n) to O(m).

4 Results

Our experimental evaluation of the prototype system was carried out on the a commodity testbed with 98 nodes. Each node is a dual Intel Pentium III CPU clocked at 500 MHz, equipped with 512 MB memory. All the nodes are connected by a Fast Ethernet switch and fabric. We split the cluster conceptually into 90 compute nodes and used the remaining 8 nodes for our I/O servers (all of which serve meta data as well). The

servers were attached to a 9 GB Quantum Atlas SCSI disk connected to an LSI Logic SCSI storage controller and use an ext3 formatted file system (writeback mode) for the storage space. All nodes run the Debian 3.1 distribution, a modified version of PVFS2 1.5.1 and Linux 2.6.16 kernel (modifications for the system call interfaces and hooks). Files are striped with a stripe size of 16 KBytes and make use of all servers to store file data. MPICH2 1.0.4p1 [16] drives our experimental evaluations involving parallel jobs. For each of the system call categories, we report results obtained with micro-benchmarks that exercise that feature alone.

4.1 Shared File Descriptors

In this micro-benchmark, we measure the costs of obtaining a file descriptor using the group open (`openg, openfh`) system calls and compare it with the independent `open` system call costs. Note that the costs of obtaining a file descriptor from the group open system calls must include the time it takes to broadcast (sockets, MPI messages, etc.) the group handle obtained from `openg`. The micro-benchmark is a parallel MPI program in which all tasks open the *same file* (using either the group open system calls or the regular open system call) and close it. The tasks of the parallel application synchronize before and after each call to obtain the file descriptor (both `openg` and `open`). In the group open system call case, the managing process (rank 0) issues an `openg` system call, obtains the group handle and calls *MPI_Bcast* to broadcast it to all the tasks. Subsequently, all tasks issue an `openfh` system call to obtain a file descriptor. Therefore, we measure the time it takes to get a file descriptor as the sum of the time it takes to do an `openg`, followed by an *MPI_Bcast* and the maximum time for an `openfh` over all the processes.

Figure 3 compares the time (in milliseconds) it takes to open (create) a file using the two schemes with varying number of processes in the MPI program and for varying pathname depths (0 and 8). Increasing the number of simultaneous processes opening the same file with `open` causes a single server to be a centralized bottleneck, while increasing the pathname depth causes multiple round-trip messages that perform lookups and attribute retrieval. Consequently, both of these parameters are important in determining the scalability of concurrent file open operations. In Figure 3, the first 4 bars (for any data point on the x-axis) show the time taken if the file was created and the last 4 bars show the time taken to open an existing file. Within the 4 bars for each category, the time taken by `openg` and `open` for a shallow path (opening/creating a file in the top-level hierarchy) and a deep path (opening/creating a file with level 8 from the root) is shown. We can make the following observations from this experiment:

- As expected, creates are slower than opening an existing file. However, costs of independent creates (for both shallow and deep files) scale poorly, while the group creates of shallow and deep files remain almost constant. The cost of independent creates increases with increasing number of clients because requests from all clients end up hitting the servers (*lookup, getattr* and *create*). In the case of the group creates, only requests from the manager process (rank 0) are sent to the file servers to create the file. Thereafter, all other processes open the file without contacting the servers. This demonstrates that group creates scale

as well as the underlying broadcast implementation (a good broadcast implementation scales logarithmically to the number of client tasks). With 90 clients, independent creates are slower than group creates by as much as a factor of 6 for both shallow and deep pathnames.

- Along similar lines, independent opens also show poor scaling with increasing number of clients in comparison to group opens. With 90 clients, independent opens are slower than group opens by as much as a factor of 15 (for shallow files) and 12 (for deep files). Since there are fewer disk I/Os involved in opening an existing file, performance improvements for group opens are more dramatic. Further, performance improvements for deeper files are less than that obtained with shallow files because of increased network round-trips and disk I/Os incurred by both schemes for looking up the file and retrieving attributes.

- For a fewer number of clients, performance of independent operations are comparable to group operations and hence we expect that higher-level libraries that make use of the group open system calls will use it judiciously. For example, if the broadcast algorithm is not optimized or if the inter-client interconnection network is slower than the client-storage interconnection network, performance of group open system calls may be slower or comparable to independent opens.

Figure 4shows a split up of the time incurred in performing the group opens and creates. A bulk of the time is incurred in the `openg` system call, since that involves multiple network round trips and disk I/Os. Since the `openfh` implementation in PVFS2 is carried out completely locally, its cost is constant and constitutes a negligible overhead. As expected, increasing the number of client tasks increases the broadcast overhead but this is still insignificant to the costs of file system related communication. Figure 4 also indicates that the relative proportion of the broadcast overhead for create operation is lower than the open operation, since creates are more I/O-bound than opening an existing file.

4.2 Noncontiguous I/O

The workload used to measure the noncontiguous I/O extensions is a parallel MPI program that determines the aggregate I/O bandwidth with varying block sizes and file region counts. Each process opens a file, and performs a set of writes, followed by a set of reads to the file, with a specific block size and number of contiguous file regions. Each parallel run consisted of 90 processes, synchronized before every IO operation. The number of memory regions in our tests remains constant at 25. A separate run is done for each set of I/O system calls available: `read/write`, `readv/writev`, and `readx/writex`. We vary the number of contiguous file regions over the range 1 to 16, while the sizes of the regions (the block size) are varied from 256KB to 4MB. In order to perform the same write and read patterns for `read/write`, `readv/writev` and `readx/writex`, the test must iterate over the region count and seek to the start of the next region before calling `read`, `write`, `readv`, or `writev`. In the `read` and `write` cases, our test iterates over each of the 25 memory regions as well. In the case

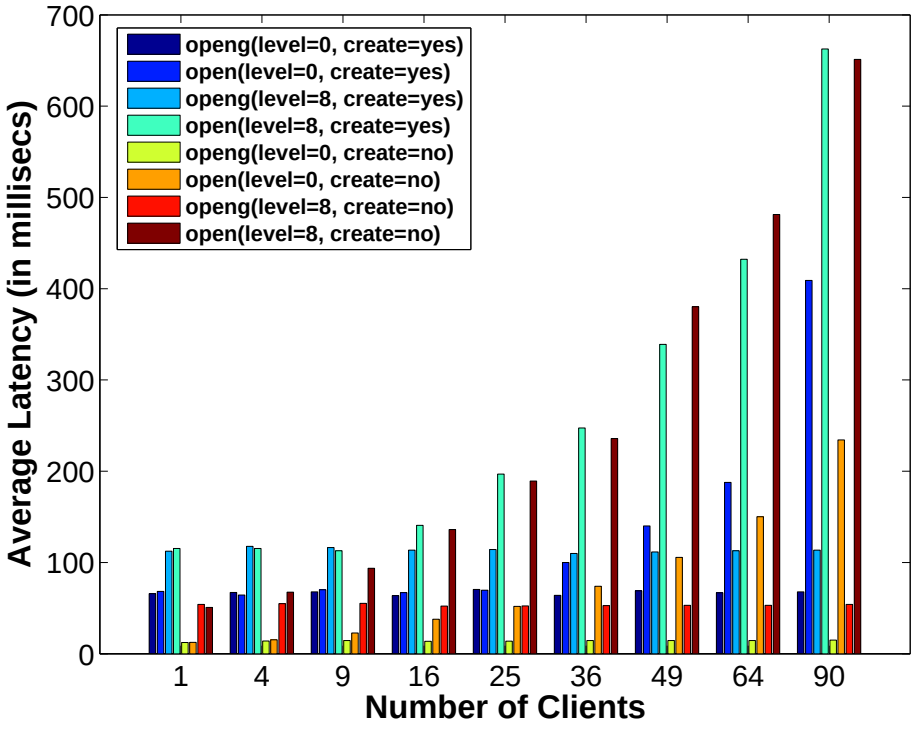

Figure 3: Time (in msecs) taken to open a file and obtain a file descriptor using either the regular `open` system call or using the `openg/openfh` system calls with varying number of client tasks and different path depths.

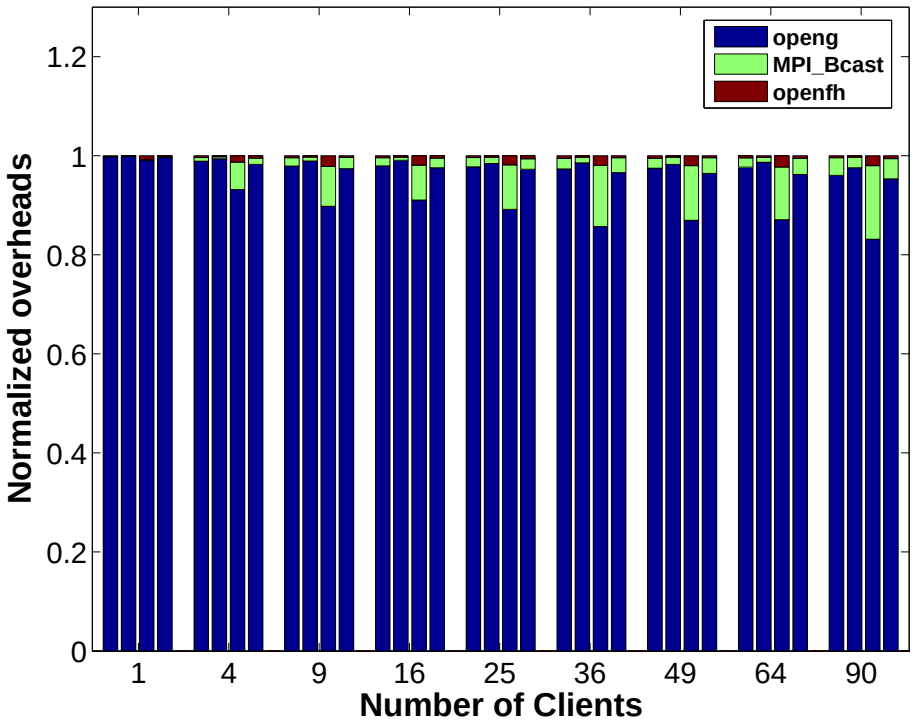

Figure 4: Breakdown/split up of the group open/create system calls (normalized overhead of *openg, bcast and openfh*). From left to right, the 4 bars are (level=0, create=yes), (level=8, create=yes), (level=0, create=no), (level=8, create=yes).

of readx/writex, since the regions are encoded in the xtvec passed to the call, only one call is made for all regions at once.

Recall that the seek operations do not incur network or disk overheads, so our test should be a direct comparison of making multiple read, write, readv or writev calls for each file region versus making one readx or writex call.

We compare the performance of the existing POSIX interfaces (read/write and readv/writev) with that of the new readx/writex extensions. The graphs in Figure 5 and 6 demonstrate the gains from using readx/writex as the number of file regions (or streams) increases. Due to caching effects at the disk level, the behavior of reads for smaller block sizes was inconsistent in our results. All the test cases that make only one round-trip (writev(stream=1), readv(stream=1), writex(stream=1,4,16), readx(stream=1,4,16)) all behave primarily the same, reaching the peak bandwidth achieved among all tests. The other test cases degrade in performance with the number of round-trips, reaching the lowest bandwidth in the write and read cases, where many more round-trips are being made.

While not all applications can benefit from using readx and writex, many access patterns in high performance computing make I/O requests with patterns that consist of many regularly strided file segments (vectored I/O). Although we demonstrate results with only 16 stream segments, we expect performance gaps to widen with larger stream counts.

4.3 Metadata

In this micro-benchmark, we measure the time it takes to retrieve all or a subset of attributes of all directory entries from a given directory using the proposed bulk metadata (getdents_plus and getdents_plus_lite) and lazy metadata attribute retrieval system calls (statlite, lstatlite, fstatlite). Given a directory name, there are multiple ways in which we could retrieve the attributes of all directory entries. The first technique (which is the traditional way) involves issuing a sequence of getdents system calls and lstat system calls on each directory entry to fetch their attributes. If only a subset of attributes are requested, we could issue lstatlite system calls on each directory entry obtained from getdents and avoid unnecessary messages to the data servers (if for instance the size attribute is not requested). This forms the process underlying the second technique. The third technique involves issuing a sequence of getdents_plus system calls which not only returns directory entries but also their attributes in a single shot. Likewise, if all attributes are not requested, we could issue a sequence of getdents_plus_lite system calls which returns directory entries and only their requested attributes. Recall that in PVFS2, file size is the only lazy attribute that can be requested by the caller.

Table 1 shows the time (in seconds) taken to read through a given directory and retrieve the attributes of all the directory entries using the 4 techniques described above. The rows of Table 1 indicate the time taken with varying number of objects (files) inside the directory. As expected, the getdents_plus system call implementation is faster than the naive getdents implementation since it retrieves both directory entries and object attributes in one shot. Further, the lazy attribute retrieval interfaces (getdents_plus_lite and stat_lite) also perform better than their counter-

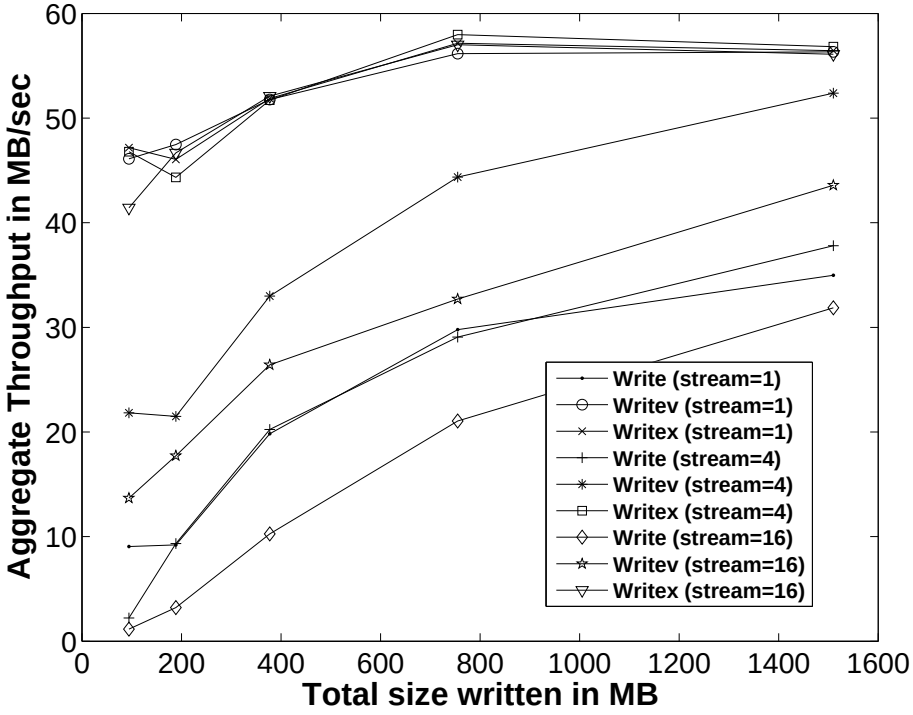

Figure 5: Noncontiguous I/O Strategies: Aggregate Write bandwidth

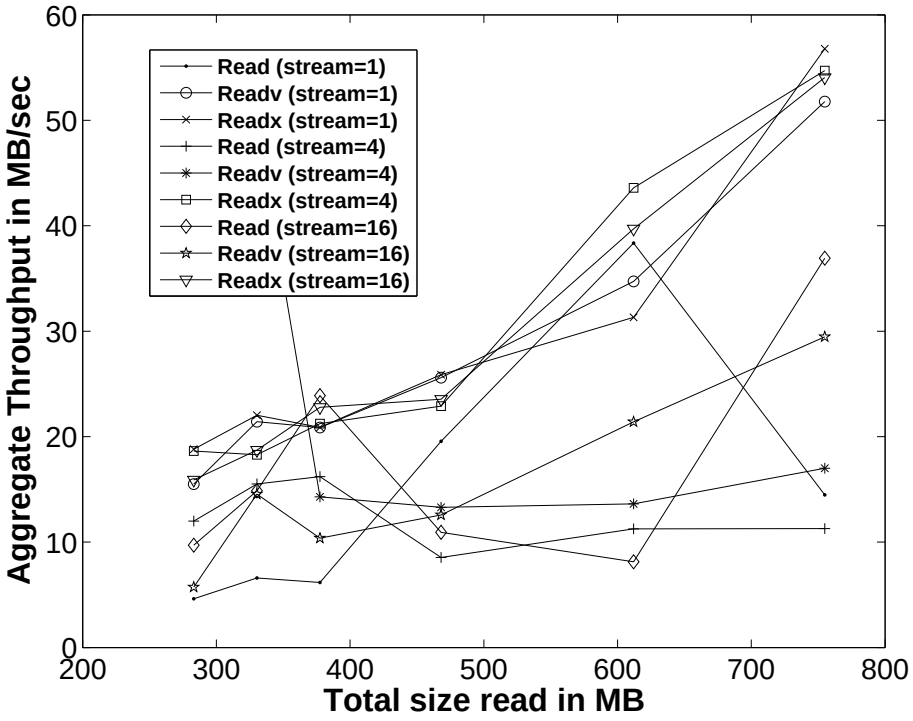

Figure 6: Noncontiguous I/O Strategies: Aggregate Read bandwidth

System call ⟶ Number of objects ↓	getdents + lstat (all attributes)	getdents + lstat_lite (all attributes but size)	getdents_plus (all attributes)	getdents_plus_lite (all attributes but size)
10	0.25	0.18	0.05	0.06
100	2.01	1.75	0.32	0.28
250	5.06	4.28	0.74	0.63
500	9.53	7.94	1.42	1.22
1000	20.79	17.18	2.83	2.45
5000	100.68	84.18	13.90	12.08

Table 1: Time taken to read through a specified number of directory entries and retrieve all or a subset of the attributes of the file system objects (in seconds).

System call ⟶ Number of objects ↓	stat (all attributes)	stat_lite (all attributes but size)
10	0.05	0.01
100	0.49	0.15
250	1.22	0.40
500	2.46	0.82
1000	4.91	1.62
5000	24.59	8.02

Table 2: Time taken to retrieve all or a subset of the attributes of a specified number of file system objects (in seconds).

parts (`getdents_plus` and `stat`) because of reduction in unnecessary messages to servers. For a 5000 entry directory, `getdents_plus` achieves a factor of 7 reduction in retrieving all the object attributes compared to the `getdents` implementation. An important thing to note here is that the `getdents_plus` implementation scales well with increasing number of objects because it makes better use of the network and I/O resources.

For the same setup, Table 2 shows the time (in seconds) taken to perform the `stat` and `stat_lite` operation only. As expected the time taken to retrieve a subset of attributes is lower than the time taken to retrieve all attributes (including the file size in case of PVFS2). Results shown in Tables 1 and 2 corroborate that bulk and lazy metadata interfaces perform orders of magnitude better than current generation metadata retrieval interfaces.

Although, this paper focuses on results obtained with micro-benchmarks, we expect that real HPC applications and tools will make use of one or more of these system call interfaces and can achieve better performance than what the micro-benchmark trends indicate.

5 Related Work

Authors in [20] provide a good overview of some of the I/O challenges and requirements faced by scientific applications and environments. Based on their observations, they also outline a set of file system design principles and management policies. A recent workshop study [18] also summarized some of the challenges faced in high-end computing environments and suggested modifications required to scale parallel file systems for next-generation scientific applications. This is a step towards standardizing mechanisms to achieve scalable metadata and data performance on large-scale parallel file systems.

Many of the ideas and concepts presented in this paper stem from work done in the context of higher-level application library interfaces (such as MPI-I/O [21], UPC-I/O [4] etc.) MPI-I/O provides interfaces for collective operations and noncontiguous accesses from which the group open and noncontiguous interfaces are derived. The UPC programming language provides a natural interface for programmers to express SPMD programs so that they can be executed on scalable supercomputers. The UPC-I/O interfaces are collective (group) functions that are invoked by all participating threads (processes). This gives the underlying implementations a chance to reorder or aggregate requests that need to be sent to the file servers. However, lack of widespread deployment and ubiquitous use of MPI-I/O (or UPC-I/O) make it harder for applications to realize any performance improvements offered by parallel file systems. Consequently, it is important that interface improvements are made available to a wider application base by incorporating them into the POSIX standards. Adding interfaces to the POSIX standard also has the added benefits of improving portability across multiple platforms and vendors. The bulk metadata interfaces for directory read operations (`getdents_plus`) is motivated by NFS Version 3 (*readdirplus* request) [17] that allows directory entries and their attributes to be read in one shot instead of requiring multiple round-trips.

Work has also been done in the context of parallel I/O libraries and file systems to enable application developers to expose hints, layout information and access patterns. However, many if not all these extensions were ad-hoc and specific to a vendor or academic prototype. Designers of the Galley parallel file system showed that many applications in scientific computing environments exhibit highly regular, but non-consecutive I/O access patterns in [15, 14]. Further, since the conventional interfaces do not provide an efficient method of describing these patterns, they present three extensions to the I/O interface to support such applications with regular access patterns. Authors in [9] present the design and implementation of Clusterfile parallel file system that enable applications to control and match I/O access patterns with file data layout.

6 Conclusions and Future Work

In this paper, we have presented the rationale, design and reference implementation of a subset of POSIX I/O system call interface extensions for a popularly used cluster parallel file system. These interface extensions address oft-cited performance bottlenecks (both for metadata and data) that prevent parallel and distributed file systems from scaling. Performance results of our prototype system using micro-benchmarks on a medium-sized cluster indicates that these extensions address the stated goals of improved performance and scalability.

Interface extensions such as those presented in this paper are by no means complete nor expected to be the only enabler of performance and scalability. We anticipate that optimizations at all levels of the file system and storage stack are needed to enable high-end scientific applications to scale. We also anticipate that newer system call interfaces may be required on parallel file systems that run on current and future generation high-end machines such as the IBM BlueGene/L and BlueGene/P. Such machines impose a unique set of requirements and challenges that cluster environments do not have and may necessitate a revisit of the POSIX I/O interfaces and wire-protocol requests. For example, I/O forwarding from designated nodes on behalf of a set of compute nodes on the BG/L machine calls for aggregation of I/O and metadata requests at such nodes as well as interfaces for compound operations. All of the above are beyond the scope of the work discussed in this paper and will be a part of future study.

Although these interfaces can be used directly in the end-user programs and utilities, we expect that the biggest users of these interfaces will be middleware and high-level I/O libraries. In particular, we expect that the proposed group open system calls and the non-contiguous I/O extensions could be leveraged by the MPICH2 ROMIO library (that implements the MPI I/O specifications) for scalable collective file opens.

Source code, documentation for the kernel patches, hooks into the PVFS2 file system, test programs and instructions for installation are available at [2].

Acknowledgments

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.

References

[1] FLASH I/O benchmark. http://flash.uchicago.edu/∼zingale/flash benchmark io/.

[2] Technical Report and PVFS2 Posix Extensions Source Tree. *Reference and URL omitted for anonymity*.

[3] P. H. Carns, W. B. Ligon III, R. Ross, and R. Thakur. PVFS: A Parallel File System for Linux Clusters. In *Proceedings of the 4^{th} Annual Linux Showcase and Conference*, 2000.

[4] T. El-Ghazawi, F. Cantonne, P. Saha, R. Thakur, R. Ross, and D. Bonachea. UPC-

IO: A Parallel I/O API for UPC v1.0. http://upc.gwu.edu/docs/UPC-IOv1.0.pdf, July 2004.

[5] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, and H. Tufo. FLASH: An Adaptive Mesh Hydrodynamics Code for Modelling Astrophysical Thermonuclear Flashes. *Astrophysical Journal Supplement*, 131(273), 2000.

[6] G. Grider, L. Ward, G. Gibson, R. Ross, R. Haskin, and B. Welch. POSIX I/O Extensions Workshop, Carnegie Mellon University, 2005.

[7] V. Henson and A. V. de Ven. Chunkfs... or how you can use divide-and-conquer to keep fsck times in bound, 2006. http://www.fenrus.org/chunkfs.txt.

[8] IEEE/ANSI Standard. 1003.1 Portable Operating System Interface (POSIX)-Part 1: System Application Program Interface (API) [C Language], 1996.

[9] F. Isaila and W. F. Tichy. Clusterfile: A flexible physical layout parallel file system. In *Proceedings of the 3^{r}d IEEE International Conference on Cluster Computing*, page 37, Washington, DC, USA, 2001. IEEE Computer Society.

[10] M. Kaczmarski, T. Jiang, and D. A. Pease. Beyond backup toward storage management. *IBM Systems Journal*, 42(2):322–337, 2003.

[11] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, 1991.

[12] D. Kotz. Disk-directed I/O for MIMD multiprocessors. *ACM Transactions on Computer Systems*, 15(1):41–74, 1997.

[13] R. Latham, N. Miller, R. Ross, and P. Carns. A Next-Generation Parallel File System for Linux Clusters. *LinuxWorld*, 2(1), January 2004.

[14] N. Nieuwejaar and D. Kotz. Low-level Interfaces for High-level Parallel I/O. Technical report, Hanover, NH, USA, 1995.

[15] N. Nieuwejaar and D. Kotz. The Galley Parallel File System. In *Proceedings of the 10th International Conference on Supercomputing*, pages 374–381, New York, NY, USA, 1996. ACM Press.

[16] M. Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, 1997. http://www.mpi-forum.org/docs.

[17] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS Version 3: Design and Implementation. In *USENIX Summer*, 1994.

[18] R. Ross, E. Felix, B. Loewe, L. Ward, G. Grider, and R. Hill. HPC File Systems and Scalable I/O. In *HEC-IWG File Systems and I/O R&D Workshop*, 2005.

[19] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the Summer 1985 USENIX Conference*, 1985.

[20] E. Smirni, R. A. Aydt, A. A. Chen, and D. A. Reed. I/O Requirements of Scientific Applications: An Evolutionary View. In *Proceedings of the High Performance Distributed Computing (HPDC '96)*, page 49, Washington, DC, USA, 1996. IEEE Computer Society.

[21] 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*, pages 23–32, New York, NY, USA, 1999. ACM Press.

[22] R. Thakur, E. Lusk, and W. Gropp. Users Guide for ROMIO: A High-

Performance, Portable MPI-IO Implementation. Technical Report ANL/MCS–TM–234, Argonne National Labs, 1997.

[23] M. Vilayannur, R. Ross, P. H. Carns, R. Thakur, A. Sivasubramaniam, and M. T. Kandemir. On the Performance of the POSIX I/O Interface to PVFS. In *12^{th} Euromicro Workshop on Parallel, Distributed and Network-Based Processing (PDP 2004)*, pages 332–339, 2004.