MERCURY: a Transparent Guided I/O Framework for High Performance I/O Stacks

Giuseppe Congiu*, Matthias Grawinkel†, Federico Padua†, James Morse*, Tim Süß†, André Brinkmann†
*Emerging Technology Group Seagate Systems Ltd Havant, United Kingdom
Email: {giuseppe.congiu, james.s.morse}@seagate.com
†Zentrum für Datenverarbeitung Johannes Gutenberg-Universität Mainz, Germany
Email: {grawinkel, padua, t.suess, brinkman}@uni-mainz.de

***Abstract*—The performance gap between processing and I/O represents a serious scalability limitation for scientific applications running on high-end computing clusters. Parallel file systems often provide mechanisms that allow programmers to disclose their I/O pattern knowledge to the lower layers of the I/O stack through a hints API. This information can be used by the file system to boost the application performance, for example, through data prefetching. Unfortunately, programmers rarely make use of these features, missing the opportunity to exploit the full potential of the storage system.**

In this paper we propose MERCURY, a transparent guided I/O framework able to optimize file I/O patterns in scientific applications, allowing users and administrators to directly control the I/O behavior of their applications without modifying them. This is done by exploiting the hints API provided by the back-end file system to guide data prefetching. The required prefetching information is annotated in a configuration file using a generic syntax and afterwards transparently translated into the corresponding file system subroutine calls. We demonstrate that MERCURY is effective in converting numerous small read requests into a few larger requests and that by doing so, it increases the I/O bandwidth, reduces the number of I/O requests reaching the back-end storage devices and ultimately the application running time. Moreover, we also propose a Linux kernel modification that allows network file systems, specifically Lustre, to work with our guided I/O framework through the posix_fadvise interface.

I. Introduction

The gap between hard disk drives (HDDs) performance and processors computing power, better known as I/O performance gap problem, represents a serious scalability limitation especially for scientific applications running on High End Computing (HEC) clusters. Parallel File Systems (PFSs) such as Lustre [9], PVFS [12] and GPFS [20], just to mention a few, try to bridge this gap by striping files across multiple storage devices and providing multiple parallel data paths to increase the aggregate I/O bandwidth and the number of IOPS. The ROMIO middleware[1] extends the POSIX I/O interface typically provided by PFSs with a richer parallel I/O interface, and through the Abstract Device I/O (ADIO) driver [22] enables transparent file access optimizations based on two-phase I/O and data sieving to adapt I/O patterns to the characteristics of the underlying file system [23] [26] [19]. Nevertheless, as Carns et al. have pointed out in their study [11] most of the scientific applications running on big clusters today still use the POSIX I/O interface to access their data.

Currently there is no available solution to overcome limitations caused by non-optimal file I/O patterns generated by applications, except to re-write them. In this context, the Linux kernel provides users with the capability to communicate access pattern information to the local file system through the `posix_fadvise()` [6] system call. The file system can use this information to improve page cache efficiency, for example, by prefetching (or releasing) data that will (or will not) be required soon in the future or by disabling read-ahead in the case of random read patterns. However, `posix_fadvise()` is barely used in practice and has intrinsic limitations that discourage its employment in real applications.

The two most used PFSs in HEC clusters nowadays, IBM GPFS and Lustre, are both POSIX compliant. Nevertheless, neither of them support the POSIX advice mechanism previously described. GPFS compensates for the lack of POSIX advice support through a hints API that users can access by linking their programs against a service library. Hints are passed to GPFS through the `gpfs_fcntl()` [2] function and can be used to guide prefetching (or releasing) of file blocks in the page pool[2]. However, unlike POSIX advice, GPFS hints can be discarded by the file system if certain requirements are not met. Lustre, on the other hand, does not provide any client side mechanism similar to GPFS hints or POSIX advice. Recently a new Lustre advice mechanism has been proposed by DDN during the Lustre User Group 2014 (LUG14) in Miami [4]. The difference in the DDN approach is that it provides control over the storage servers (OSSs) cache instead of the file system client cache.

In this paper we propose and evaluate a novel guided I/O framework called MERCURY [5] able to optimize file access patterns at run-time through data prefetching using available hint mechanisms. MERCURY communicates file I/O pattern information to the file system on behalf of running applications using a dedicated process that we call *Advice Manager*. In every node of the cluster, processes can access their files using an *Assisted I/O library* that transparently forwards intercepted requests to the local *Advice Manager*. This uses `posix_fadvise()` and `gpfs_fcntl()` to prefetch (or release) data into (or from) the client's file system data cache. The *Assisted I/O library* controls for which files advice or hints should be given, while the *Advice Manager* controls how much data to prefetch (or release) from each file. Monitored file paths and prefetching information are contained in a configuration file that can be generated either manually or automatically

[1]Implementation of MPI I/O specifications from Argonne National Laboratory included in MPICH package (http://www.mpich.org/).

[2]GPFS pinned memory used for file system caching.

once the I/O behaviour of the target application is known. The configuration file mechanism allows us to decouple the specific hints API provided by the back-end file system from the generic interface exposed to the final user thus making our solution portable.

With this approach we are able to generate POSIX advice and GPFS hints for applications that do not use them but can receive a benefit from their use. We accomplish this asynchronously, with very low overhead, and without any modification of the original application. We demonstrate that our approach is effective in improving the I/O bandwidth, reducing the number of I/O requests and reducing the execution time of a 'ROOT' [3] [7] big data science based analytic application.

Additionally, we propose and evaluate a modification to the Linux kernel that makes it possible for Lustre and in principle other networked file systems to participate in activity triggered by the `posix_fadvise()` system call, thus allowing it to take advantage of our guided I/O framework benefits.

The remainder of this paper is organised as follows. Section II covers the background on file systems support to guided I/O interfaces (POSIX advice and GPFS hints), Section III presents concept, design and implementation of the MERCURY prototype highlighting the main contributions of the work. This section also describes the kernel modifications that enable POSIX advice on Lustre, Section IV presents the evaluation of our prototype on three file systems: a local Linux ext4 file system, a GPFS file system and a Lustre file system, Section V presents related works on data prefetching, and finally Section VI presents conclusion and future work.

II. Background on Guided I/O Interfaces

In this section we describe in detail the POSIX advice provided by the Linux kernel as well as the GPFS hints. Some of the specifics presented in this section will be useful to understand our design choices explored in Section III.

A. The POSIX Advice API

The Linux kernel allows users to control page cache functionalities through the `posix_fadvise()` system call:

int *posix_fadvise(**int** fd, **off_t** offset, **off_t** len, **int** advice)*

This system call takes four input parameters: a valid file descriptor representing an open file, starting offset and length of the file region the advice will apply to, and finally the type of advice. The implementation provides five different types of advice, that reflect different aspects of caching.

The first two advice in Table I have an impact on spatial locality of elements of the cache. `POSIX_FADV_SEQUENTIAL` can be used to advise the kernel that a file will be accessed sequentially. As result the kernel will double the maximum read-ahead window size in order to have a greedier read-ahead algorithm. `POSIX_FADV_RANDOM`, on the other hand, can be used when a file is accessed randomly and has the effect of completely disabling read-ahead, therefore only ever reading the requested data. Finally, `POSIX_FADV_NORMAL` can be used to cancel the previous two advice-messages and reset the read-ahead algorithm to its defaults. These three advice types apply to the whole file, the offset and length parameters are ignored for these 'modes'.

TABLE I: Values for *advice* in the *posix_fadvise()* system call

Advice	Description
`POSIX_FADV_SEQUENTIAL`	file access pattern is sequential
`POSIX_FADV_RANDOM`	file access pattern is random
`POSIX_FADV_NORMAL`	reset file access pattern to normal
`POSIX_FADV_WILLNEED`	a file region will be needed
`POSIX_FADV_DONTNEED`	a file region will not be needed
`POSIX_FADV_NOREUSE`	file is read once (not implemented)

Two of the remaining three advice types have an impact on the temporal locality of cache elements. `POSIX_FADV_WILLNEED` can be used to advise the kernel that the defined file region will be accessed soon, and therefore the kernel should prefetch the data and make it available in the page cache. `POSIX_FADV_DONTNEED` has the opposite effect, making the kernel release the specified file region from the cache, on the condition that the corresponding pages are clean (dirty pages are not released). Finally, the implementation for `POSIX_FADV_NOREUSE` is not provided in the kernel.

One important aspect of `posix_fadvise()` is that it is a synchronous system call. This means that every time an application invokes it, it blocks and returns only after the triggered read-ahead operations have completed. This represents a big limitation especially if we consider `POSIX_FADV_WILLNEED` that may need to prefetch an arbitrarily large chunk of data. In this scenario the application may be idle for a long period of time while the data is being retrieved by the file system.

B. The GPFS Hints API

Similarly to POSIX advice, GPFS provides users with the ability to control page pool functions through the `gpfs_fcntl()` subroutine:

int *gpfs_fcntl(**int** fileDesc, **void*** *fcntlArgP)*

The subroutine takes two inputs: the file descriptor of the open file that hints will be applied to, and a pointer to a data structure residing in the application's address space. The indicated data structure contains all the information regarding what hints should be sent to GPFS. Specific hints are described by means of additional data structures that are contained in the main struct. Table II summarizes all the available hints data structures and reports the corresponding description for each of them.

Hints are not mandatory and GPFS can decide to accept or ignore them depending on specific conditions. Let us consider the multiple access range hint as an example (`gpfsMultipleAccessRange_t` in table II). The data structure corresponding to this hint is reported in Listing 1.

[3]Data analysis framework developed at CERN.

TABLE II: Data structures provided by GPFS to describe different hints

Hint data structure	Description
`gpfsAccessRange_t`	defines a region of the file that needs to be accessed
`gpfsFreeRange_t`	defines a region of the file that needs to be released
`gpfsMultipleAccessRange_t`	defines multiple regions of the file that needs to be accessed
`gpfsClearFileCache_t`	releases all the page pool buffers held by a certain file

```
#define GPFS_MAX_RANGE_COUNT 8
#define MAX_RANGE_COUNT GPFS_MAX_RANGE_COUNT

typedef struct
{
    int               structLen;
    int               structType;
    int               accRangeCnt;
    int               relRangeCnt;
    gpfsRangeArray_t accRangeArray[MAX_RANGE_COUNT];
    gpfsRangeArray_t relRangeArray[MAX_RANGE_COUNT];

} gpfsMultipleAccessRange_t;
```

Listing 1: Multiple access range hint data structure

`gpfsMultipleAccessRange_t` contains two range arrays instead of just one: `accRangeArray`, used to define `accRangeCnt` blocks of the file that GPFS has to prefetch, and `relRangeArray` used to define `relRangeCnt` blocks of the file previously requested using `accRangeArray` and that are no longer needed. Unlike posix_fadvise the user has to manage the list of blocks for which hints have been sent, updating whether they are still needed. Indeed, if the accessed blocks are not released, GPFS will stop accepting new hints once the maximum internal number of prefetch requests has been reached.

III. Concept & Design

The first part of this section presents the concept, design and the implementation of the MERCURY prototype. The second part describes the Linux kernel modifications that allow Lustre to work with our solution through the posix_fadvise interface.

The I/O software stack of MERCURY is depicted in Figure 1. Besides the standard I/O libraries we add two software components, an *Assisted I/O library* (AIO), used to intercept I/O calls issued by applications and an *Advice Manager* (AM) process that receives messages sent from the *Assisted I/O library* and generates POSIX advice and GPFS hints. The library is preloaded by the runtime linker before other libraries through the `LD_PRELOAD` mechanism and uses UNIX domain sockets to communicate with the *Advice Manager*. In the case of GPFS hints *libgpfs* provides the correct hints API to the *Advice Manager*, other file systems will use the posix_fadvise syscall.

The proposed architecture adds two major contributions. First of all, it allows us to use the Linux advice API as well as the GPFS hints API asynchronously through the *Advice Manager*. This means that we can effectively overlap I/O and computation phases in target applications. Secondly, it enables us to generate POSIX advice and GPFS hints transparently, without the need to modify the application. The information required by the *Advice Manager* is extracted from observations of the application's I/O behaviour during a preliminary run and then written to a configuration file to be used in following runs.

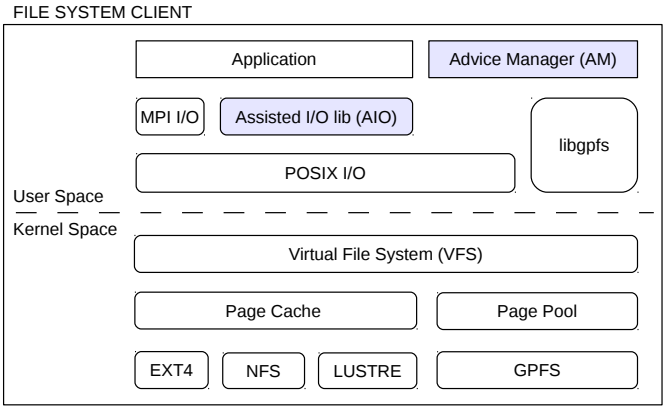

Fig. 1: MERCURY I/O software stack. *Assisted I/O library* and *Advice Manager* communicate through UNIX domain sockets. Data can reside in a local Linux file system, in Lustre or in GPFS.

In the rest of this section we describe in detail the different aspects of our design including the interprocess communication between the two software entities and the prefetching request generation using the `posix_fadvise()` system call or the `gpfs_fcntl()` function.

A. Interprocess Communication

We now describe in detail how interprocess communication is implemented and how messages sent from the *Assisted I/O library* are handled by the *Advice Manager*. Figure 2 depicts the architecture of the two software components introduced by our design. The *Advice Manager* is made up of three smaller modules: a *Request Manager* (RM) that receives requests sent by the *Assisted I/O library*, a *Register Log* (RL) that keeps track of which files are currently handled by the *Advice Manager*, and an *Advisor Thread* (AT) that receives read requests from the *Request Manager* through a queue and issues POSIX advice and GPFS hints.

In order to enable asynchronous prefetching we delegate the task of sending synchronous hints or advice to the *Advice Manager*. When an application issues an open call for a file, the *Assisted I/O library* intercepts it and sends a message to the *Advice Manager*. The message contains a string of the form: "**Register** *pid pathname fd*", plus additional ancillary information explained later. This string tells the *Request Manager* to register the pid of the process opening the file with pathname and file descriptor number, in the register log. As a consequence the *Request Manager* performs two operations, first it asks the *Request Log* to register the new file. From this point on, future read calls for the file will be monitored by the *Advice Manager*. Second, it creates a new

Advisor Thread that will take care of generating POSIX advice or GPFS hints depending on which file system the file resides in. I/O calls coming from the application are never blocked by the *Assisted I/O library*. The reason is that the *Advice Manager* can become congested by too many requests coming from different processes and we do not want to reflect this on the behaviour of the application.

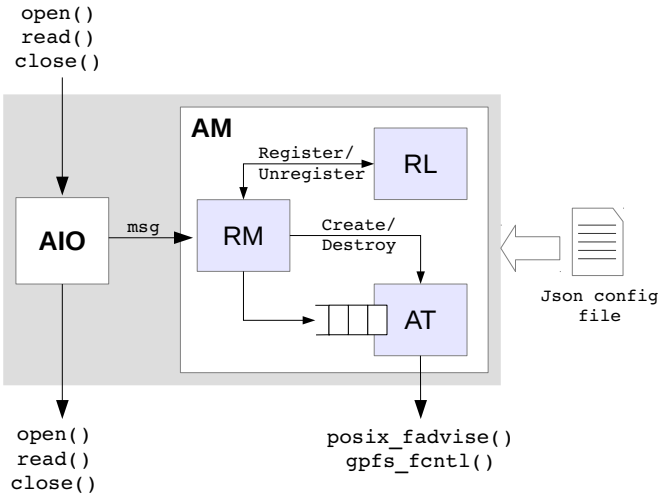

Fig. 2: Detailed architecture for the *Advice Manager* (AM) component. This can be further divided into three blocks: *Request Manager* (RM), *Register Log* (RL), and *Advisor Thread* (AT).

Both POSIX advice and GPFS hints affect an open file, identified by its file descriptor number. For the *Advice Manager* to send advice or hints on behalf of the application, it needs to share the open file with the application. When sending messages from the *Assisted I/O library* to the *Advice Manager* we use `sendmsg()`. Besides normal data, this system call allows the transfer of ancillary information. One use of such information is to send a remote process a 'file descriptor' [21] via a UNIX domain socket [8]. These numbers are just an index into the kernel's list of a process's open files. When sending a file descriptor using `sendmsg()`, the kernel copies a new reference to the open file descriptor, and adds it to the receiving process's open files list. The *Advice Manager* receives a new file descriptor number, (which will likely be different to the number sent), which points to a file descriptor shared with the application. This allows us to send hints or advice for the shared file.

B. File Data Prefetching

POSIX advice and GPFS hints are issued using the *Advisor Thread* created by the *Request Manager* during the register operation (Figure 2). When an application performs a read operation for an open file, the *Assisted I/O library* sends to the *Advice Manager* a message containing a string of the form: **"Read** *pid fd off len*". This string includes the pid of the process, the application's file descriptor number for the file, the offset within the file and the length of the request. The pid and the file descriptor number are used by the *Request Manager* module only to identify the corresponding *Advisor Thread*. When the correct thread has been identified the *Request Manager* pushes the offset and the length of the read request into a queue. This queue is accessed by the *Advisor Thread* that uses the read information to trigger prefetch requests using the local file descriptor and keeps track of all the prefetched data using a block cache data structure.

The *Advisor Thread* uses `posix_fadvise()` and `gpfs_fcntl()` to generate prefetch requests for the underlying file systems (Figure 2). For files residing in local file systems and Lustre, the `POSIX_FADV_WILLNEED` advice from Table I is used to bring the data into the kernel page cache. For files residing in GPFS the `accRangeArray` in the `gpfsMultipleAccessRange_t` data structure in Listing 1 is used to define which blocks of the file should be brought into the GPFS internal cache (page pool). The size of the file regions to prefetch is defined in a Json[4] configuration file, loaded at startup by both the *Advice Manager* and the *Assisted I/O library*. This is the only point of configuration for the user and it contains, besides other information, a list of files and directories that the *Assisted I/O library* should monitor. An example configuration file is shown below.

```
{
    "File": [
    {
        "Path": "/path/to/target/file",
        "BlockSize": 4194304,
        "CacheSize": 8,
        "ReadAheadSize": 4,
        "WillNeed": [
        {
            "Offset": 0,
            "Length": 0
        }]
    }],
    "Directory": [
    {
        "Path": "/path/to/target/dir",
        "Random": [
        {
            "Offset": 0,
            "Length": 0
        }]
    }]
}
```

Listing 2: Example of Json configuration file.

As can be seen in Listing 2 the structure of the configuration file is very simple. It allows users to define which files POSIX advice or GPFS hints should be applied to by setting the 'Path' field to the full file path and the regions of the file that are likely to be accessed in terms of offset and length. In the case of POSIX advice users can also define directories to which a global advice should be applied (e.g. randomly accessed files in the directory). Additionally, when indicating a 'WillNeed' advice users can directly control the caching behaviour of the *Advisor Thread* block cache. In particular, they can define the granularity of the prefetch request ('BlockSize'), how many blocks can be fitted into the *Advisor Thread* cache ('CacheSize') and how many blocks of data should be read ahead starting from the current accessed block ('ReadAheadSize'). Clearly the example in Listing 2 is not exhaustive. More complex configuration files can be generated by administrators

[4]Open standard format that uses human-readable text to transmit data objects consisting of attribute-value pairs (http://www.rfc-editor.org/rfc/rfc7159.txt).

(or automatic tools) to dynamically change the I/O patterns of applications in order to best adapt them to the underlying storage system.

The replacement policy for the block cache in the *Advisor Thread* uses a LRU algorithm. In order to prefetch data, the open file is divided into blocks of size 'BlockSize' and entire blocks are loaded/released into/from memory as the application progresses. In the case of GPFS the `accRangeArray` hint is used to prefetch up to 'ReadAheadSize' blocks ahead starting from the block touched by the current request. When the number of blocks in the cache has reached 'CacheSize', if more blocks are requested, older blocks will be released using the `relRangeArray` hint to make space for the new ones. In the case of POSIX advice, the behaviour is the same but blocks are loaded into memory using the `POSIX_FADV_WILLNEED` advice and released using the `POSIX_FADV_DONTNEED` advice. The hints interface is automatically selected by the *Advice Manager* at runtime depending on the file system hosting the target file.

The *Advisor Thread* block cache also provides a very basic level of coordination among processes accessing the same file. In fact, different *Advisor Thread* instances hinting the same file on behalf of different processes share the same block cache. Blocks requested by one process will appear in the block cache and future accesses to those blocks by other processes will not trigger new prefetching requests.

In general the configuration file can be used to describe any of the advice listed in Table I and the hints listed in Table II. To define a new scenario, we may consider a file region accessed sequentially for which the `POSIX_FADV_SEQUENTIAL` advice type could be used, and another region accessed randomly for which the `POSIX_FADV_RANDOM` advice type could be used. In this case, the configuration file would contain a list of file regions, specifying which type of advice messages are suitable. The right advice will be selected according to which part of the file is being accessed currently. This feature allows us to overcome another limitation of the Linux advice implementation that has been mentioned in Section II-A, namely, the first three advice types apply to the whole file since the implementation in the kernel completely disregards the byte ranges specified by the user.

Finally, when the application closes the file the *Assisted I/O library* sends to the *Advice Manager* a message containing a string of the form: **"Unregister** *pid fd*". This string includes the pid of the process and the file descriptor number of the file to be closed. In response to this request the *Request Manager* tells the *Register Log* to unregister the file and destroys the *Advisor Thread*, it also closes its shared copy of the file.

C. POSIX Advice integration with Lustre

Lustre is a high performance parallel file system for Linux clusters. It works in kernel space and takes advantage of the available page cache infrastructure. Additionally, it extends POSIX read and write operations with distributed locks to provide data consistency across the whole cluster. Even though Lustre makes use of the Linux kernel page cache, the previously described POSIX advice syscall has no effect on Lustre. The reason can be understood by looking at Figure 3. This reports the simplified call graph for the Lustre read operation in the file system client. To simplify the explanation, the figure is divided into four quadrants. Along the x-axis we have the native kernel functions (e.g. `generic_file_aio_read`), separated by the Lustre specific functions (e.g. `lustre_generic_file_read`). Along the y-axis we have page operations (e.g. `find_get_page`) separated by the file operations (e.g. `generic_file_aio_read`).

We can notice that Lustre extends the kernel code with additional file and page operations through the Lustre Lite component. These are the functions used by the kernel to fill the file operations table and the address space operations table. The `posix_fadvise()` system call in the kernel translates into `fadvise64()`. In the case of `POSIX_FADV_WILLNEED` this function directly invokes `force_page_cache_readahead()` which has no effect on `ll_readpage()`. Other advice such as `POSIX_FADV_{NORMAL,SEQUENTIAL,RANDOM}` are disabled in Lustre by setting the kernel read-ahead window size to zero. This is done so that lustre will not speculatively try to gain a highly-contended lock to fulfil an optimistic read-ahead request.

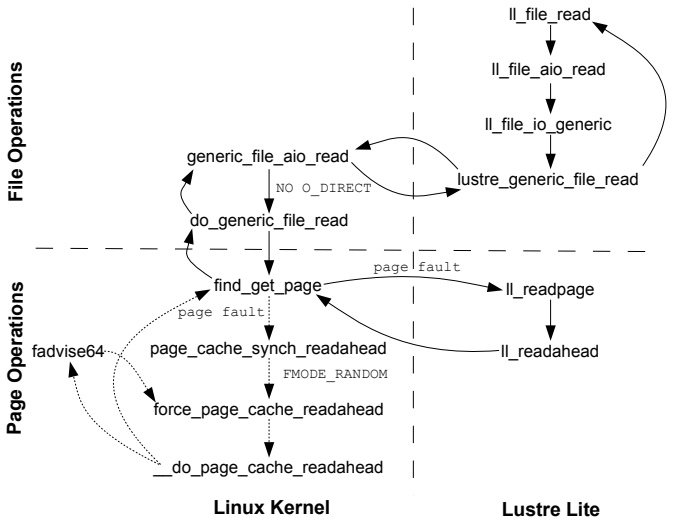

Fig. 3: Simplified function call graph for the read operation in Lustre. For page operations in the Linux kernel the picture also shows the call graph typically followed by local reads as well as the call graph for the `POSIX_FADV_WILLNEED` advice in the `posix_fadvise()` implementation (dashed line).

In order to enable `POSIX_FADV_WILLNEED` in Lustre we modified the call graph of `fadvise64()` presented in Figure 3 to invoke the `aio_read()` operation in the file operations table for the open file and block until all the data has been read into the page cache. In this way we can force the kernel to invoke the corresponding file read operation in Lustre, acquiring locks as appropriate. Of course this mechanism still works with local file systems which eventually will end up calling `force_page_cache_readahead()` as in the original version.

To prevent the new generated read from altering the read-ahead state of normal read operations, in `fadvise64()` we

create a new `struct file` using the `dentry_open()` routine and set the access mode flag (`f_mode`) of the new file to `FMODE_RANDOM` (which is exactly what the `POSIX_FADV_RANDOM` advice message does to disable read-ahead for random accessed files). This mechanism works perfectly with local file systems but has no effect on Lustre's read-ahead algorithm which is independent from the Linux kernel read-ahead. Therefore, `POSIX_FADV_WILLNEED` in the case of Lustre prefetches a bit more data than requested. This is acceptable for now but a future implementation will also modify the Lustre code to make sure the behaviour is the same in both cases.

Finally, our kernel patch does not require any user buffer to be provided with the new read operation. To avoid data being copied to user space we pass a null pointer to the `aio_read()` routine. Additionally we defined a new `ki_flag` for the kernel I/O control block (`kiocb`), that we called `KIF_FORCE_READ_AHEAD`. This new flag is checked in the `generic_file_aio_read()` routine and if set the `do_generic_file_read()` routine is invoked with a pointer to the `file_read_actor_dummy()` routine. `file_read_actor()` is normally the routine responsible for copying the data from the page cache to the user space buffer. Since in our case there is no user space buffer, the dummy routine just returns success.

IV. Evaluation

We now present the evaluation of our *Assisted I/O library* and *Advice Manager* prototypes with a real world application used by physicists at the data processing center of the University of Mainz (ZDV). Our testbed is composed by a test cluster of seven nodes, mainly intended to evaluate the proposed Linux kernel modification with the Lustre file system. We start with a concise description of the system as well as a detailed analysis of the target application's I/O pattern, and then present the results of our experiments.

We evaluate the performance of our framework using two metrics, the execution time of the test application and the number of reads completed by every target file system.

A. Test Cluster

As already mentioned, this small cluster is aimed mainly to test our modified kernel with Lustre. The reason is that it was not possible to disrupt the production cluster, affecting hundreds of users, by re-installing the operating system kernel. In order to make realistic comparisons between Lustre and GPFS, the test cluster also has a GPFS file system on comparable hardware. Both filesystems have a single disk server each, one Dell R710 acts as GPFS network shared disk (NSD) server and another as Lustre object storage server (OSS). The R710 are equipped with two quadcore E5620 @ 2.4GHz and 24GB main memory. For storage, both disk servers share a MD3200 array with 2 controllers and 4 MD1200 expansion shelves for a total of 60 2TB drives. The Storage is formatted in 4 15 dynamic disk pools. This is the LSI/Netapp type of declustered RAID, which distributes the 8+2 RAID6 stripes evenly over all 15 disks for better rebuild performance. The disk block size is set to 128KiB, which results in a RAID Stripe size of 1MiB. The four disk pools are then split on the Array into LUNs, one of the LUNs from each disk pool is then used for GPFS and another one from each pool is used for Lustre. This results in comparable resources for both filesystems and tests do not interfere with each other, as long as only one filesystem is tested at a time. While the GPFS filesystem embeds the metadata with the data, Lustre needs a separate Metadata Server (MDS). This is hosted by a SuperMicro server equipped with one quadcore Xeon E3-1230 @3.3GHz and 16GB of main memory, as metadata target (MDT) it uses a 120GB SSD Intel 520. Four other machines of the same type, equipped with an eight core E3-1230 @3.3GHz processor and 16GB of main memory, work as compute nodes and file system clients. All machines, servers and clients, are equipped with Intel X520DA 10Gigabit adapters and connected to a SuperMicro SSE-X24S 24 ports 10 Gigabit switch. Both, the GPFS and Lustre file systems are formatted with a block size of 4MB.

B. Real World Application

Our target real world application is written using 'ROOT', an object-oriented framework widely adopted in the experimental high energy physics community to build software for data analysis. The application analyzes data read from an input file in the 'ROOT' format (structured file format).

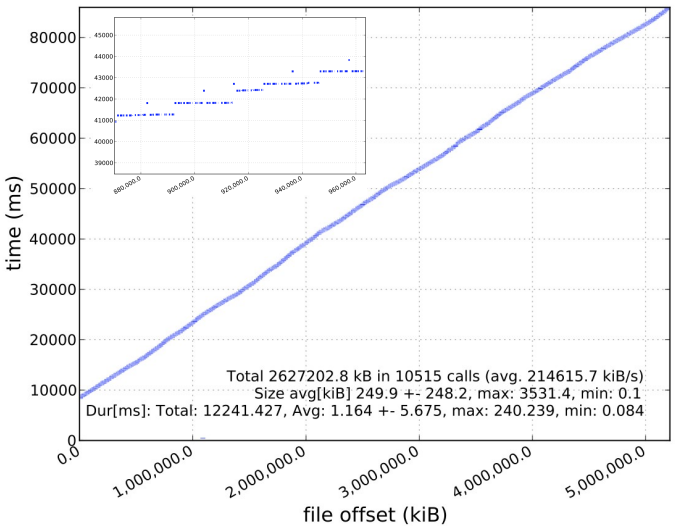

Fig. 4: I/O read profile of the target application under analysis extracted from the the GPFS file system in the test cluster.

First of all we characterized the application's I/O pattern for a target file using traces and statistics extracted through several tools such as *strace*, *ioapps* [3] and GPFS's *mmpmon* [1] monitoring tool. Figure 4 shows the I/O pattern along with some additional statistics. As it can be seen, in this specific case (5GB file), the application issues a total of 10515 `read()` system calls to read about 2.6GB of total data. The average request size is 250kiB and the time spent waiting for I/O is 12 seconds, when running on the test cluster.

At a first glance the general I/O behaviour of the application looks linear, most of the accesses to the file follow an increasing offset. Nevertheless, adjacent reads are separated by gaps (a strided read pattern). In a few cases this gap becomes negative, meaning that the application is moving backwards in

the file to read some data previously left behind (as reported in the rectangular insertion in Figure 4).

After a detailed I/O pattern analysis we could divide the target file into contiguous non overlapping ranges. Within these ranges reads happen to have increasing offset. Even though the general I/O pattern of the application for different files looks similar[5], the size of the non overlapping ranges may change significantly. This general behaviour can be modelled using a configuration file in which a 'WillNeed' hint covers the whole file from beginning to end (i.e. 'Offset' and 'Length' equal to 0). The backwards seeks can be accounted for using the 'CacheSize' parameter to keep previously accessed blocks in cache. In this way we effectively emulate a sliding window that tracks the application's I/O behaviour. This would not be possible by just using a, e.g., `POSIX_FADV_WILLNEED` advice on the whole file before starting the application like shown by Figure 5. The reason is that if the file size is equal or smaller than the cache size, we would have a large number of valuable pages discarded from the cache to load data that will be accessed at the end of the application. Additionally, if the file size is bigger than the cache size we would have the file system discarding blocks at the beginning of the file as the blocks at the end are preloaded, effectively forcing the application to access these blocks from the I/O servers instead of the cache. With our approach, on the other hand, we keep in the cache only a small, controlled number of blocks (the ones currently accessed), while the older blocks are discarded since no longer needed.

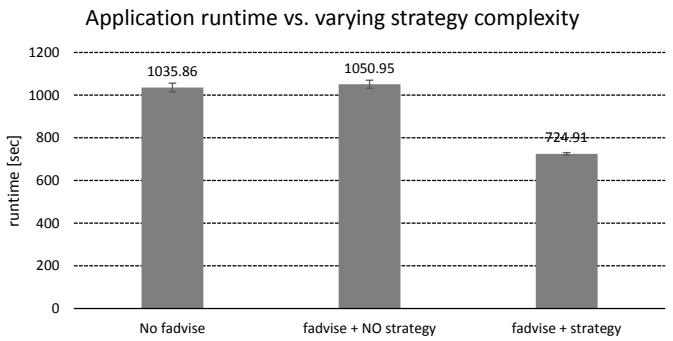

Fig. 5: Comparison between different usage stategies of posix_fadvise for an input file of 55GB residing in an ext4 file system. The first bar represents the case in which no advice is used, the second bar represents the case in which a POSIX_FADV_WILLNEED is issued for the whole file at the beginning of the application and the third bar represents the case in which POSIX_FADV_WILLNEED is issued using MERCURY.

To assess the impact of our prototype on the application and file systems performance we considered the application execution time and the number of reads accounted for by the respective file systems. We conducted our experiments without file system hints and then with file system hints issued transparently to the application by the *Advice Manager*. Furthermore, we ran each experiment three times and calculated average, minima and maxima for each metric. In order to avoid caching affecting our measurements, extra care was taken to clean all the relevant caches for the different file systems. For ext4 and Lustre this was accomplished by using the command line:

$$echo\ 3 > /proc/sys/vm/drop\_caches$$

on the file system clients. Additionally, for Lustre this command was also executed on the OSS to avoid the server side cache to be retained. In the case of GPFS, the file system client's page pool was cleaned using the clean file cache hint in Table II, the NSD servers do not cache any data.

[5]Due to space limits we do not report the comparison between different files.

C. Execution Time

To measure the performance improvements that our prototype can deliver to the application's runtime we conducted two set of tests. In the first test we varied the size of the input file from 5 to 95GB. This is mainly aimed to study the behaviour of the 'ROOT' application using different input file sizes and how our solution behaves when the file becomes bigger than the available cache space. In the second test we varied the number of 'ROOT' instances running simultaneously from 1 to 8. By doing so we study the interaction of multiple processes accessing the file system and how these can benefit from the prefetching hints generated by MERCURY. Figure 6 reports the results for the described experiments. All the tests where performed using a 'BlockSize' of 4MB, a 'CacheSize' of 8 blocks, a 'ReadAheadSize' of 4 blocks, and a 'WillNeed' hint covering the whole file (i.e. with 'Offset' and 'Length' equal to 0), resulting in each process consuming up to 32MB of cache space and 160MB in total for 8 application instances. The 'WillNeed' on the whole file causes the *Advisor_Thread* to issue up to 4 ('ReadAheadSize') prefetching requests for blocks of 4MB sequentially, starting from the current accessed block. This has the same effect of data sieving in ROMIO, optimizing the access size and allowing the application to read the requested data randomly from the cache instead of the file system. The produced effect is particularly beneficial in the case of Lustre and ext4, as it can be seen in Figures 6a and 6c. In these cases we measure reductions in the execution time of up to 50% circa, with respect to the normal case. For GPFS we can still observe an improvement but this is more contained compared to the other file systems (Figure 6b). The reductions in the execution time measured in GPFS are on average up to 10%, with respect to the normal case. The reason is that the default prefetching strategy in GPFS works better that traditional read-ahead. In fact, by disabling the prefetching in GPFS we observe reductions in the execution time comparable to the other file systems.

As far as Figures 6d, 6e and 6f are concerned, these account for the effect of processes' concurrency on the file system. Before continuing with the discussion we have to make a note here. In our architecture, only one process per file system's client issues (through multiple *Advisor Thread*s) hints on behalf of running applications. This introduces some overhead, since we have to pass the access information from the *Assisted I/O library* to the *Advice Manager*, but has the advantage of better coordinating accesses to the same file from multiple processes. Nevertheless, we found that in the case of GPFS, despite the fact of having multiple *Advisor Thread*s, only one process among the many was receiving a benefit from the prefetching hints. The reason is that GPFS seems

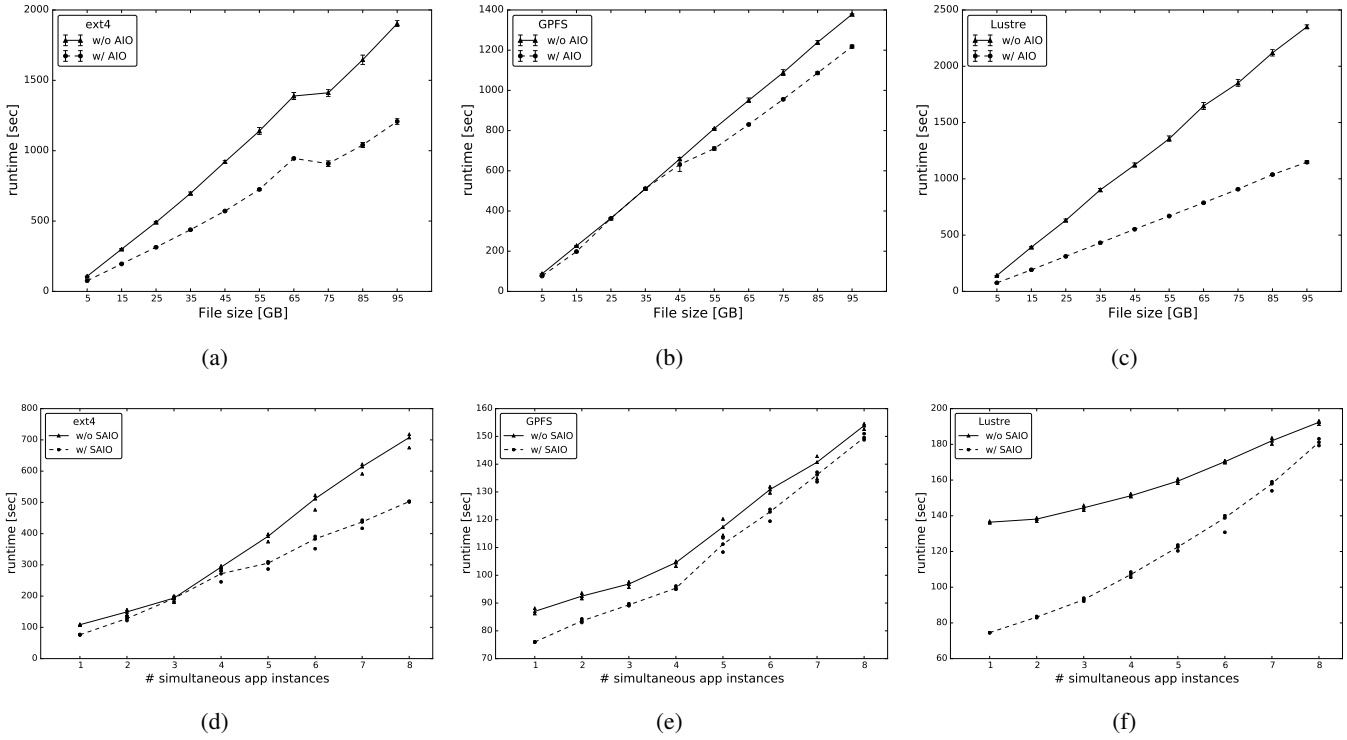

Fig. 6: Running time of the ROOT application for the three file system under study using different input file sizes (6a, 6b and 6c) and different number of instances accessing a file of 5GB (6d, 6e and 6f).

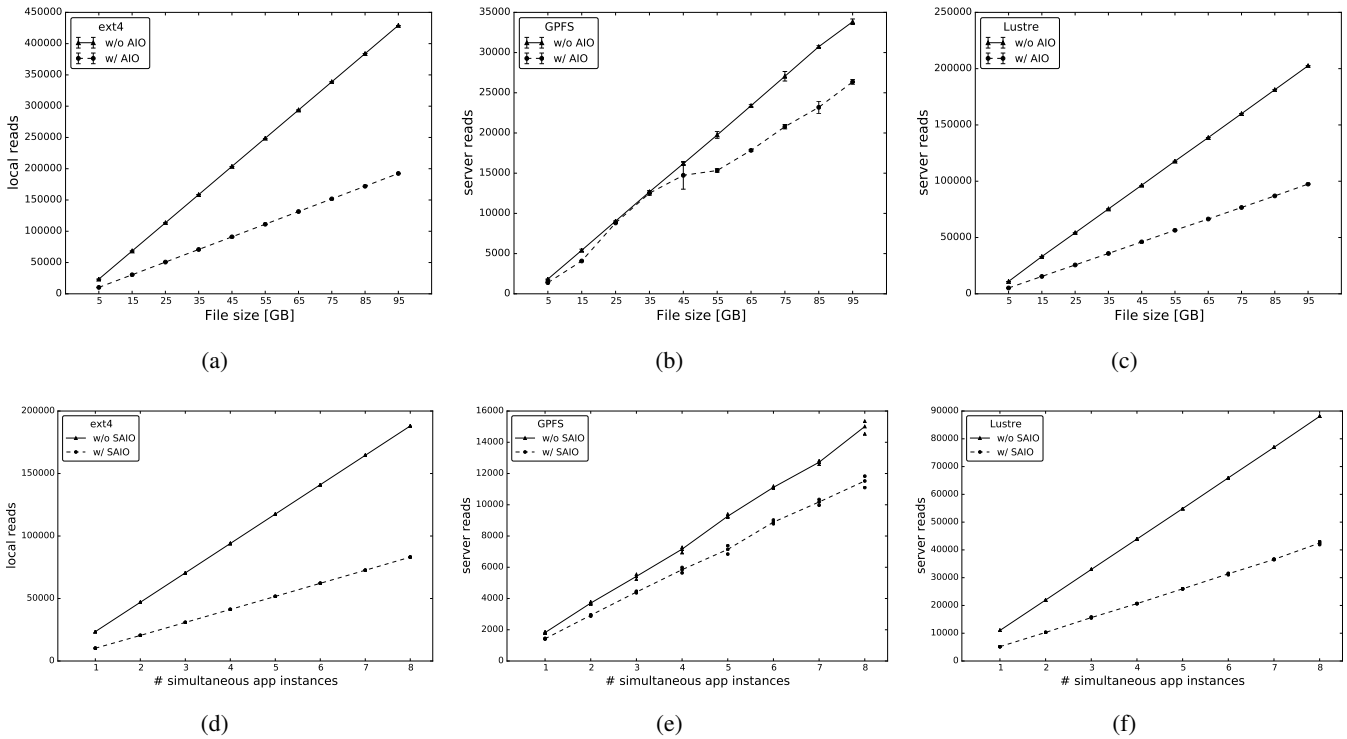

Fig. 7: Reads processed by local ext4, GPFS and Lustre I/O servers for various input file sizes (7a, 7b and 7c) and multiple instances of ROOT accessing a file of 5GB (7d, 7e and 7f).

to have the restriction of hinting only one file per process. For this reason, we developed another variant of MERCURY in which the AIO library, now renamed *Self Assisted I/O library* (SAIO), internally provides the creation and the handling of multiple *Advisor Thread*s. Looking at the figures generated with the new SAIO library we can assess the effectiveness of the prefetching hints for the three file systems considered. In particular, Lustre provides the best runtime improvements compared to the case in which no hints were used. GPFS shows a more contained improvement since the I/O time is already small compared to Lustre and ext4. Finally, ext4 can really benefit from prefetching hints especially for high process counts. Overall, excluding ext4, when we increase the number of processes the runtime improvements shrink. This is probably due to the saturation of the file system client bandwidth.

D. Read Request Rate

Figure 7a, 7b and 7c report the number of read requests accounted for by the different file systems under study. In the specific, the figures show how the number of reads at the I/O server side for both GPFS and Lustre can be substantially reduced with our approach. This has a significant impact in HPC cluster in which the file system may be accessed by many thousand of processes at the same time. Reducing the number of requests for an application can increase the number of IOPS available for others. This result is also confirmed for multiple instances of the 'ROOT' application running concurrently (Figure 7d, 7e and 7f).

V. Related Work

In the past researchers have tried to alleviate the I/O performance gap problem by analyzing I/O patterns and exploiting their knowledge to guide I/O using, for example, data prefetching. Tran and Reed [24] presented an automatic time series modelling and prediction framework for adaptive I/O prefetching, named TsModeler. They combined ARIMA and Markov models to describe temporal and spatial behaviour of I/O patterns at file block level. TsModeler was integrated with the experimental file system PPFS2 and used to predict future accesses. He et al. [16] proposed a pattern detection algorithm, based on the sliding window algorithm in LZ77 as base for building Markov models of I/O patterns at file block level. The model was afterwards used by a FUSE based file system to carry out prefetching. Chang and Gibson [13], unlike previous works, did not build mathematical models but instead used speculative execution of the application code to guide data prefetching.

Other works tried to bring the same idea to higher level I/O libraries such as MPI I/O, HDF5 or PnetCDF to take advantage of the richer semantic, data dependencies and layout information. Chen, Byna, Sun, Thakur and Gropp [14] proposed a pre-execution based prefetching approach to mask I/O latency. They provided every MPI process with a thread that runs in parallel and takes responsibility for prefetching future required data. Prefetching in the parallel thread was enabled via speculative execution of the main process code. The same authors in [10] proposed to exploit parallel prefetching using a client-side, thread based, collective prefetching cache layer for MPI I/O. The cache layer used I/O pattern information, in the form of I/O signatures, together with run time I/O information to predict future accesses. Chen and Roth [15] took inspiration from the collective I/O optimization enabled by ROMIO to design a collective I/O data prefetching mechanism that exploited global I/O knowledge. He, Sun and Thakur [17] proposed to analyze high level data dependencies exposed in PnetCDF, accumulate this knowledge building data dependency graphs and finally use them to perform prefetching.

VanDeBogart, Frost and Kohler have previously used the Linux advice API to build a prefetching library [25] for programmers to use. Prost et al. integrated the GPFS hint functionalities in the ROMIO ADIO driver for GPFS [18]. In this context they exploit data type semantic in file views to prefetch parts of the file that will be soon accessed.

In contrast to previous works, we do the following things differently. We do not try to automatically build mathematical models of I/O patterns and use them to accurately generate prefetching requests nor do we speculatively execute the application binary. In fact, we believe that users and administrators have the best understanding about the applications and their systems, and can exploit their knowledge and expertise to improve the storage system performance. We demonstrate that experienced users with a deep knowledge of their applications I/O behavior can convert non-optimal I/O patterns, in particular small random reads, into patterns that can be adapted to the underlying file system characteristics, and therefore give optimal performance. Furthermore, previously described approaches are not suitable for small random read patterns since they rely on accurate knowledge of I/O behaviour to prefetch every single request one after the other. This still degrades the storage system performance due to the large number of I/O requests and seek operations hitting the storage devices. On the other hand, by using the POSIX advice and GPFS hints APIs, we can prefetch the region of the file that will be accessed and filter random requests using the cache.

In this work we focus on providing the infrastructure that enables users to access file system specific interfaces for guided I/O without modifying applications and hiding the intrinsic complexity that such interfaces introduce.

VI. Conclusion & Future Work

In this paper we presented a guided I/O framework prototype that can be used to set POSIX advice and GPFS hints on behalf of applications transparently to the user. This is done by adding annotations regarding which regions of a file to prefetch into a configuration file that is afterwards fed to the *Assisted I/O library* and *Advice Manager* modules.

We focused predominantly on read patterns which characterize a class of scientific applications known as big data science analytics. These class of applications differ from HPC applications in the type of I/O pattern they use. Indeed, HPC applications are dominated by writing of large amounts of checkpoint data to a shared (N-1 pattern) or multiple files (N-N pattern) for restart purposes or, more generally, for post processing. Big data analytics applications, on the other hand, read large amounts of input data for processing and write very little volumes of output data (results) to the file system. Currently, there is a convergence of these two paradigms that brings big data analytics applications to run on high-end computing clusters, typically as post processing phase for data generated

by HPC applications such as, e.g., climate and earthquake simulations. Here we considered 'ROOT' as representative for big data science analytics and we demonstrated that by using the hints API provided by the file system in an appropriate way it is possible to improve the storage system usage and ultimately the application performance.

In the future we plan to further explore the problem of big data science analytics applications in HPC, studying more profusely the different types of available I/O patterns and how hints can be used to improve the cache utilization efficiency and thus applications' performance.

ACKNOWLEDGMENT

This work has been supported by the Marie Curie Initial Training Networks (MCITN) of the European Commission (contract no. 238808), by the Exascale IO (E10) initiative and by the DFG TRR 146: Multiscale Simulation Methods for Soft Matter Systems.

REFERENCES

[1] GPFS monitoring tool. https://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp?topic=%2Fcom.ibm.cluster.gpfs.v3r5.gpfs200.doc%2Fbl1adv_mmpmonch.htm.

[2] gpfs_fcntl subroutine. https://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp?topic=%2Fcom.ibm.cluster.gpfs.v3r5.gpfs100.doc%2Fbl1adm_fcntl.htm.

[3] ioapps I/O profiling tools. https://code.google.com/p/ioapps.

[4] Lustre file system acceleration using server or storage-side caching: basic approaches and application use cases. http://opensfs.org/wp-content/uploads/2014/04/D2_S27_LustreFileSystemAccelerationUsingServerorStorageSideCaching.pdf.

[5] MERCURY guided I/O framework. https://github.com/gcongiu/mercury.git.

[6] posix_fadvise. http://linux.die.net/man/2/posix_fadvise.

[7] ROOT, A Data Analysis Framework. http://root.cern.ch/drupal.

[8] UNIX Domain Sockets. http://linux.die.net/man/7/unix.

[9] P. J. Braam, "Lustre: a scalable high-performance file system," White Paper, November 2002.

[10] S. Byna, Y. Chen, X. he Sun, and R. Thakur, "Parallel i/o prefetching using mpi file caching and i/o signatures."

[11] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and improving computational science storage access through continuous characterization," in *Proc. of the 27th IEEE Conference on Mass Storage Systems and Technologies (MSST)*, 2011, pp. 1–14.

[12] P. H. Carns, W. B. Ligon, III, R. B. Ross, and R. Thakur, "Pvfs: A parallel file system for linux clusters," in *Proc. of the 4th Annual Linux Showcase and Conference*. USENIX Association, pp. 317–327.

[13] F. Chang and G. A. Gibson, "Automatic i/o hint generation through speculative execution," in *Proc. of the 3rd Conference on Operating Systems Design and Implementation (OSDI)*, ser. OSDI '99. Berkeley, CA, USA: USENIX Association, 1999, pp. 1–14.

[14] Y. Chen, S. Byna, X.-H. Sun, R. Thakur, and W. Gropp, "Hiding i/o latency with pre-execution prefetching for parallel applications," in *Proc. of the ACM/IEEE Conference on Supercomputing (SC)*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 40:1–40:10.

[15] Y. Chen and P. Roth, "Collective prefetching for parallel i/o systems," in *Proc. of the 5th Petascale Data Storage Workshop (PDSW)*, 2010, pp. 1–5.

[16] J. He, J. Bent, A. Torres, G. Grider, G. Gibson, C. Maltzahn, and X.-H. Sun, "I/o acceleration with pattern detection," in *Proc. of the 22nd*, ser. HPDC '13. New York, NY, USA: ACM, 2013, pp. 25–36.

[17] J. He, X.-H. Sun, and R. Thakur, "Knowac: I/o prefetch via accumulated knowledge," in *Proc. of the IEEE International Conference on Cluster Computing (Cluster)*, 2012, pp. 429–437.

[18] J.-P. Prost, R. Treumann, R. Hedges, B. Jia, and A. Koniges, "Mpi-io/gpfs, an optimized implementation of mpi-io on top of gpfs," in *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, ser. SC '01. New York, NY, USA: ACM, 2001, pp. 17–17. [Online]. Available: http://doi.acm.org/10.1145/582034.582051

[19] J.-P. Prost, R. Treumann, R. Hedges, A. Koniges, and A. White, "Towards a high-performance implementation of mpi-io on top of gpfs," in *Euro-Par 2000 Parallel Processing*, ser. Lecture Notes in Computer Science, A. Bode, T. Ludwig, W. Karl, and R. Wismller, Eds. Springer Berlin Heidelberg, 2000, vol. 1900, pp. 1253–1262. [Online]. Available: http://dx.doi.org/10.1007/3-540-44520-X_177

[20] F. Schmuck and R. Haskin, "Gpfs: A shared-disk file system for large computing clusters," in *Proc. of the 2002 USENIX Conference on File and Storage Technologies (FAST)*, 2002, pp. 231–244.

[21] W. R. Stevens and S. A. Rago, *Advanced programming in the UNIX environment*. Addison-Wesley professional computing series, May 2013, ch. Advanced IPC, pp. 642–652.

[22] R. Thakur, W. Gropp, and E. Lusk, "An abstract-device interface for implementing portable parallel-i/o interfaces," in *Proc. of the 6th IEEE International Symposium on The Frontiers of Massively Paralle Computation*. IEEE Computer Society Press, 1996, pp. 180–187.

[23] ——, "Data sieving and collective i/o in romio," in *Proc. of the 7th IEEE International Symposium on The Frontiers of Massively Paralle Computation*, ser. FRONTIERS '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 182–.

[24] N. Tran and D. A. Reed, "Automatic arima time series modeling for adaptive i/o prefetching," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 4, pp. 362–377, Apr. 2004.

[25] S. VanDeBogart, C. Frost, and E. Kohler, "Reducing seek overhead with application-directed prefetching," in *Proceedings of the 2009 conference on USENIX Annual technical conference*, ser. USENIX'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 24–24.

[26] L. Ying, "Lustre ADIO collective write driver," White Paper, October 2008.